IMPLEMENTATION AND PERFORMANCE ANALYSIS OF A PARALLELIZED PARTICLE FILTER ALGORITHM

By

Tanmay Misra

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Electrical Engineering

May, 2014

Nashville, Tennessee

Approved:

Professor Gabor Karsai

Professor William H. Robinson

ACKNOWLEDGEMENTS

This thesis would not have been possible without the immense patience and support that my advisor Dr. Gabor Karsai showed during the entire course of this project, and I would like to begin by thanking him. Each of our discussions led to the next step of this research and helped me think about the non-obvious and question the obvious. The greatest thing that I have learned from Dr. Karsai is how to do scientific research and how to focus on the important questions. He has been a great advisor, and I shall always remain indebted to him.

Besides my advisor, I would like to thank Dr. William H. Robinson, my second reader and co-advisor, for all his inputs and insights regarding the experiments and the project. His suggestions in preparing this manuscript were wonderful. I am especially grateful to him for his constant encouragement in the initial days of my graduate career. I would also like to thank Dr. Richard Alan Peters for helping me understand the FastSLAM algorithm and Dr. Mitch Wilkes for his insights on probabilistic robotics. I would like to acknowledge the financial, academic and technical support of the EECS department and the Graduate School at Vanderbilt University and the entire Vanderbilt community.

I am very much thankful to Trey Reece, doctoral student at Vanderbilt University, for his comments and suggestions on my research and on how to present it. They were invaluable. And finally many thanks to my two friends Shweta Khare and Ashish Tapdiya, both doctoral students in Computer Science department at Vanderbilt University, for helping me understand many simple fundamentals in computer science.

Though this comes last, I am and shall always be eternally grateful to my mother and father, for their love, support and constant encouragement, and especially for receiving my calls in the middle of the night in India. When things weren't good, I made it through only because they made me keep looking forward.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Particle filter (PF) [1] is one of the most important signal filtering technique adapted in a gamut of estimation and tracking applications that includes the *Condensation algorithm* for visual tracking [2] in computer vision, Monte Carlo Localization(MCL) [3] in mobile robot localization, FastSLAM [4] for solving the simultaneous localization and mapping problem (SLAM), fault diagnosis [5] in robotics, person tracking [6] in surveillance systems, and recently in Intelligent vehicles [7]. The use of PF for navigation and positioning applications is also described in [8].

The PF estimates the state of a dynamic system from a sequence of noisy measurements made on the system [9]. The accuracy of the estimates for the state variable(s) being tracked determines the efficiency of PF algorithms. This is a function of the number of particles (point mass representations of probability densities [9]) used in the algorithm. However, increasing the number of particles also increases the computational requirements of the algorithm, thus limiting its practical use in many real-time applications as mentioned in the previous paragraph. Therefore, there is a need to find a computationally efficient implementation of PF without compromising on its accuracy.

In the last decade, the microprocessor industry has adopted a new path to increase the computing performance of a microprocessor by increasing the number of CPUs in a single processing unit [10]. Such processors are referred to as multi-core processors [11]. With computers powered with multi-core processors becoming ubiquitous, there is a tremendous interest in the software research community to develop tools [12][13][14][15] to utilize the available parallelism in hardware. Parallel programming can also improve the performance and optimization of many scientific algorithms which are inherently parallel.

This thesis presents an approach to implement a PF algorithm, namely the Monte Carlo Localization (MCL) technique used in mobile robot localization, by adopting the new parallel programming tools to design a distributed and parallelized implementation of the algorithm. The research focusses on improving the computational efficiency of the MCL and the methodology can also be applied to other similar PF algorithms. The primary aim of this study is to understand the behavior of the MCL algorithm from a computational performance point of view.

Implementation of the PF based MCL presented in this research has been developed in C# and Visual Studio 2013. The DotNet framework's parallel programming libraries [14], [16] have been used to design the parallel implementation. Parallelization has been achieved by explicitly creating and running threads in multiple CPUs concurrently. This is known as multithreading.

Multithreading (MT) [17], [18] is a type of parallel programming, in which a process is divided into separate threads, each of which can run independently. With MT, threads can execute concurrently when two or more threads are in progress at the same time. Parallelism arises when at least two threads are executing simultaneously. A multithreaded application can be executed on a single processor, in which the processor switches execution resources between threads resulting in concurrent execution. But on a multi-core processor, each thread of the multithreaded application can run on separate CPUs at the same time, resulting in parallel execution.

Designing a multithreaded application requires handling many subtle issues which can become a performance bottleneck. These issues usually don't arise in a sequential implementation. Examples are load balancing among threads, oversubscription of the multi-core processor, contention and synchronization issues, and usage of thread unsafe methods, to name a few. In this thesis, such limitations are analyzed and its influence of the parallelized implementation of the MCL is shown. They are relevant to most parallel multithreaded applications. Some mechanisms such as use of thread pools to avoid over and under subscription, static load balancing to improve parallel performance in MCL, and use of thread-local storage to reduce synchronization issues have been demonstrated.

The performance analysis of the algorithm has been done by varying the number of threads in the code, and by assigning each thread to a specific CPU. A thread can be assigned to a specific core by setting its processor affinity property to one of the CPUs in the multi-core processor. Assigning a CPU core to a thread has the benefit of eliminating CPU switching by the operating system during execution where a thread migrates from one CPU to another. Such jumps introduce additional overhead in the running-time of the thread due to continuous cache memory-reloads for each different thread running in the same CPU. Setting CPU-thread affinity also improves the data locality and reduces the cache-coherency traffic among the cores, thus optimizing cache performance.

In C#, thread affinity to CPUs can be set by the `Process.ProcessorAffinity` property. In Intel OpenMP runtime library `KMP_PLACE_THREADS` and `KMP_AFFINITY` environment variables can be used to set the thread affinity.

The MCL algorithm is a sequential and recursive execution of three steps: 1) Sampling, 2) Importance Weight Calculation, and 3) Resampling. The mathematical framework of the Sampling and Importance weight calculation step exhibits data parallelism. This parallelism was brought out in the parallelized implementation which is not possible in a sequential implementation. Parallelization of both these steps were studied in detail with varying number of threads and number of particles. In the parallelized implementation of the MCL, a maximum speed up of **5.413** over the sequential

implementation was achieved with 8 threads, tested on an *Intel Core i7-2630QM* quad-core processor. The speedup was comparable to the theoretical speedup of 5.935 predicted by Amdahl's law [19].

The organization of the remaining thesis is as follows. Chapter 2 introduces the Monte Carlo Localization method and Particle Filters. A brief overview on the Bayes Filter equations and recursive state estimation is presented. The MCL algorithm is studied is detail, including the kinematics of a mobile robot. In Chapter 3, steps for a parallel implementation of the MCL is described. The workflow of the MCL algorithm suggests the possibility of mapping the algorithm to a multi-core processor for parallel execution. Chapter 4 presents the working of the mobile robot simulator which generates the set of measurement and control data for the MCL. The behavior of the mobile simulator is based on the kinematics equations of Chapter 2. In Chapter 5 the experiments for the performance analysis of the sequential and parallel version of MCL is presented in detail, with Chapter 6 mentioning the results of the experiments in the same order. Finally, the contribution of this research is summarized in Chapter 7.

CHAPTER 2

BACKGROUND

In this chapter, the background theory on state estimation and particle filters is discussed along with the mathematical model of the Monte Carlo Localization. The work in this thesis is based on probabilistic robotics [20].

Recursive State Estimation

State Estimation is the problem of estimating and tracking the state variables of a dynamic system over time from a set of noisy measurements made on the system. In a Bayesian framework for state estimation, a probability density function is computed over all the possible values (states) of the variable of interest.

Suppose *x* represents a quantity that is being estimated based on some information or data *y*; this requires the calculation of the conditional probability distribution *p(x/y)* called the *posterior probability distribution* over state *x*, conditioned on the input data *y*. The probability *p(x)* is referred to as the *prior probability distribution*. The probabilities *p(x)* and *p(x/y)* represents the knowledge of the state of the system **before** and **after** incorporating the data *y* into the calculation, respectively. Usually the state *x* is not directly measurable. In such cases, *Bayes rule* [20] is used to infer the value of *x* from data *y* by using the *inverse probability, p(y/x)*. This can be expressed as:

$$p(x \mid y) = \eta p(y \mid x) p(x) \tag{2-1}$$

where η is the normalizer.

The data is usually classified into two types- **control data u_t** and the **measurement data z_t**.

(**Definition**) *Control Data*: Control data convey information about the change of state from time t_1 to t_2. Control data can also be control commands that are responsible for change of state.

$$u_{t1:t2} = u_{t1},\ u_{t1+1},\ u_{t1+2} \ldots u_{t2}$$

represents the sequence of control data from time t_1 to t_2 that causes the state *x* at t_1 to transition to a state at t_2. Control data at any time *t* is denoted as u_t.

(**Definition**) *Measurement Data*: Measurement data convey information about the external environment that can influence the state x_t at time t and is denoted as z_t.

$$z_{t1:\,t2} = z_{t1},\ z_{t1+1},\ z_{t1+2}\ldots z_{t2}$$

is the sequence of measurement data from time t_1 to t_2.

Following Markov's assumption [20] the generation of state x_t from previous state x_{t-1} can be expressed as

$$p(x_t \mid x_{0:t-1}, z_{1:t-1}, u_{1:t}) = p(x_t \mid x_{t-1}, u_t) \tag{2-2}$$

This equality states that if x_{t-1} is a complete state [20], then it is a sufficient summary of all previous time steps and thus can be used to stochastically generate state x_t, given control data u_t and z_t. The conditional probability: $\mathbf{p(x_t|x_{t-1},u_t)}$ is called the *state transition probability* and describes probabilistically the transition of state from x_{t-1} to x_t as a function of time and control data.

By the same Markov's assumption, the measurement data is modelled by the inequality

$$p(z_t \mid x_{0:t}, z_{1:t-1}, u_{1:t}) = p(z_t \mid x_t) \tag{2-3}$$

This equality holds true if x_t is a complete state. The probability: $\mathbf{p(z_t|x_t)}$ is called the *measurement probability*, which describes the probability of making the observation z_t from state x_t.

Bayes Filter Algorithm

The Bayes filter algorithm [20] is a recursive solution to state estimation and tracking for dynamic state systems. A Bayes filter calculates a conditional probability distribution for the estimated state(s), which is also referred to as belief of the state, written $bel(x_t)$. This is a posterior probability distribution for the state variables based on the available data, expressed as

$$bel(x_t) = p(x_t \mid z_{1:t}, u_{1:t}) \tag{2-4}$$

A Bayes Filter consist of two essential steps- *state prediction* and *state update*. The inputs to the algorithm are the prior belief $bel\ (x_{t-1})$ at time *t-1*, control data u_t, and measurement data z_t at time *t*. The first step of the algorithm is state prediction, in which a new belief is computed from prior belief and the most recent control data. It is represented by the equation,

$$\overline{bel}(x_t) = \int p(x_t \mid u_t, x_{t-1})\, bel(x_{t-1})\, d\, x_{t-1} \tag{2-5}$$

The second step is the state update which computes the measurement update as:

$$\mathrm{bel}(x_t) = \int \eta p(z_t \mid x_t)\overline{\mathrm{bel}}(x_t) \tag{2-6}$$

where η is the normalization constant.

The Bayes Filter algorithm recursively performs both the steps for each new control and measurement data. The $\overline{bel}(x_t)$ calculated at the end of the first step, predicts the state just after incorporating u_t and before z_t. The initial belief $bel(x_t)$ at time t = 0 can be uniformly distributed over the entire state space of the state *x*, if the initial value is unknown.

Particle Filters

Particle Filters are sequential Monte Carlo Simulation methods [21] that implement a recursive Bayesian filtering technique for state estimation of multi-modal beliefs. Particle filters provide an approximate solution to the integral equations of Bayes filter (equation 2.5 and 2.6). Particle filters are also known as the *Sequential Importance Sampling (SIS)* algorithm. They are widely used as a statistical technique for tracking the state of a dynamic system that can be modeled by a Bayesian network [22]. Particle filters are approximate state estimators and can be used for online Non-Linear and Non-Gaussian tracking [9] unlike Kalman Filters [23][24].

The basic idea in particle filtering is to track the state variable by representing the belief of the state, which as described previously is a conditional posterior probability distribution over the entire state space, by a set of random samples drawn from this distribution. These samples are called *particles,* written as,

$$X_t = x_t^1, x_t^2, ..., x_t^N$$

where *N* is the maximum number of particles in the particle set X_t.

Each particle x_t^n $(1 \le n \le N)$ represents a possible value of the state variable at the given time *t*. Each particle also has a weight associated with itself which is a measure of the likelihood of the state that the particle represents. The estimate of the state is based on the particle samples and their weights. The larger the weight, higher is the probability of the particle to be the correct estimate of the state. For a given large set of particle samples X_t, the weighted set of particles is a discrete approximation of the system, $p(x_t)$.

There are different versions of particle filter algorithms developed from the generic framework of the SIS algorithm. Some of the commonly used are *Sampling Importance Resampling(SIR) Filter* [1], *Auxiliary Sampling Importance Resampling (ASIR) filter* [25], *Regularized Particle filter(RPF)* [26],and *Rao-Blackwellised Particle Filtering(RBP)* [26]. The *Sampling Importance Resampling (SIR)* is a

sequential Monte Carlo method which allows for on-line tracking of state estimates. The Monte Carlo Localization (MCL) for robots is an example of the SIR filter algorithm that is being studied in this research.

The input to a SIR filter algorithm at time *t* are the particle set X_{t-1} from the previous time step, control input u_t and measurement input z_t. The output of the algorithm is the new set of particles representing the state at time *t*.

The SIR algorithm consists of the following three steps which are applied recursively in discrete time index: (1) Sampling, (2) Importance Weight calculation, and (3) Resampling.

Sampling – For each particle *n* in the particle set X_{t-1}, a new state $x_t^{[n]}$ is generated at time *t* from the previous sample of n, $x_{t-1}^{[n]}$ and from new control information u_t. This corresponds to the state prediction step (equation 2-5) of the Bayes Filter, written as

$$x_t^{[n]} \sim p(x_t \mid x_{t-1}^{[n]}, u_t) \tag{2-7}$$

for n = 1 to N. The resulting sample set is X_t.

In the sampling step, the algorithm tracks the state transition from x_{t-1} to x_t by creating a new particle for each of the possible states that *x* can transition to, at time *t*. This new set of particles represents the probability distribution over the state of the variable before incorporating the measurement data which is the belief $\overline{bel}(x_t)$. In other words, at the end of the Sampling step, a prediction of the state of the system is made. This prediction is based only upon the previous known state and the new control information. The presence of noise renders this calculation inaccurate adding uncertainty to the value of the state variable. The second step of the algorithm tries to remove the uncertainty in the predicted value by use of measurement data.

Importance Weight calculation: The output of the sampling step is a random estimate of the state of the system. It is a collection of new samples of particles. In this step, an *importance weight or Importance Factor* is calculated for each particle $x_t^{[n]}$. This step corresponds to the update step of Bayes Filter, in which the measurement data w_t is incorporated to the probability distribution. The weight associated with each particle is calculated as,

$$w_t^{[n]} \sim p(z_t \mid x_t^{[n]}) \tag{2-8}$$

The importance weight w_t for particle *n* is the likelihood of making an observation z_t from the estimated state x_t. The importance weight is a number, between 0 and 1(since this is a probability), assigned to each new particle sampled from the sampling step. The magnitude of the number indicates how close the

particle is to the true state of the system. The magnitude is closer to 1 if the particle will produce a measurement very similar or comparable to the actual measurement thus being a good estimate of the state variable. As each particle has a unique value for its state, the probabilities or weights are also unique numbers. Particles with higher weights are classified as good particles and those with lower weights are bad particles because they are poor estimates of the state.

After each particle has been assigned a weight, at this stage, a new estimation of the true state can be made from the particles with larger weights. As described in the above paragraph, the particles with lower weights are a poor representation of the actual state and hence do not contribute to the state calculation. Appropriate techniques can be used to determine the true state, for example the particle with the largest weight can be chosen to represent the true state or is the best estimate of the state. Another approach is to take the weighted sum of all states and the mean is the value of the state.

Particle filters often suffer from a problem known as the degeneracy phenomenon [9]. As the recursive SIS algorithm loops over the state prediction and state update steps, after a few iterations, a situation might come when all but one single particle has a very high weight and the rest of the particle's will have negligible weights. Then it is this particle that represents the true state estimate and all other particles are poor estimates. But because all the step of the algorithm are applied to each of the particles including the bad particles, this problem can lead to wastage of computational efforts on processing particles that will not have a meaningful contribution to state estimation.

Resampling is the third step of the SIR algorithm. The SIR algorithm is in fact the addition of the resampling step to a SIS algorithm. The resampling step reduces or avoids the degeneracy problem. The idea of resampling is to reselect a new set of samples from the sample set X_t generated by the sampling step. The selection of the particles for the new set is based on their importance weight calculated in the second step. The intention is to remove particles with lower weights and replace them by duplicate copies of the particles with higher weights. This is also known as *Sampling With Replacement* [9].

However, not all the lower weight particles are replaced, which might lead to other problems such as *particle impoverishment.* Particle impoverishment is a result of loss of diversity in the sample distribution and can often make a particle filter fail to track. To avoid throwing out all the bad particles, the particles are chosen such that the selection process is done proportional to the particle weights. In this way, it can be ensured that particles with higher weights are selected more often together with lesser number of particles of lower weight. The total number of particles always remains the same. This allows the probability distribution to be high (more particles) in high probability areas and low in low-probability areas. Also, because particles of different high weights are selected, this allows the filter to keep track of multiple,

accurate hypotheses. This is why a particle filter is considered to be an approximate solution compared to a Kalman Filter implementation where there is an exact single estimate.

A drawback of resampling is that it limits the opportunity to completely parallelize the SIR algorithm as all the particles are combined in the resampling step. Selection of a particle in this step depends on its own and other particles' weight.

Many methods for resampling have been suggested by the research community. Some commonly used techniques are stratified sampling, residual sampling [22] and systematic resampling [27].

Mobile Robot Localization

The ability of a mobile robot to perform autonomous navigation is a result of a concerted action of four complex functions - *Localization, Cognition or Path planning, Motion Control* and *Perception.*

Robot Localization refers to the problem of estimating the pose of a mobile robot relative to its environment, which is represented by a map. Localization provides the answer to the question, *Where is the robot now?* It is very important to know the exact pose or location of the robot, in order to determine its next course of action. For example, if a robot is commanded to move from, say Point A to Point B, in order to plan its path and motion the robot needs to know its current position, and its position at all the time till it reaches Point B. This ability to correctly figure out (estimate) the current position is known as localization.

(**Definition**) *Robot Pose* – The pose of a robot is defined as the coordinates of the robot in some frame of reference. Example: For a mobile robot in a two dimensional planar environment, the pose can be represented by a vector of three variables *(x, y, θ)* where *x* and *y* are the location co-ordinates and *θ* is heading direction or yaw in degrees or radians, with respect to an external axis. . Mathematically, pose is written as a vector:

$$p = \begin{Bmatrix} x \\ y \\ \theta \end{Bmatrix} \tag{2-9}$$

(**Definition**) *Maps*: A map *m* is a list of objects and their properties. Objects are known landmarks that includes physical, tangible or geometric features present in the real world in which the robot is navigating. Not all objects can be classified as features. These features have to be distinct enough for the sensor to detect them and extract meaningful information. Properties of objects can be Cartesian locations of features in a *feature based map* or an index corresponding to a specific location in a *location-based map*. A map is represented by some data structure, for example:

$$m = \{m_1, m_2, m_3 ... m_N\}$$

where *N* is the total number of objects in the environment.

In this research, a planar map is used in which each object is denoted by $m_{x,y}$ which explicitly states the world coordinate of the object in the map.

The motion of the robot can be tracked with the help of various sensors that can provide a stream of data about the robot or about the environment of the robot. These data may include range measurements

to obstacles [28][29] , compass or gyroscopes readings on robot's heading angle, odometry readings on speed and distance travelled *(dead reckoning)*. The current research is on using cameras for vision based robotic systems, by means of computer vision algorithms to extract information from camera images[30][31].

However sensor data are inaccurate. Robot motion is subject to various kinds of noise, which could be because of wheel slippage, unaccounted friction, uneven terrain, magnetic influence on compass or may be due to imperfect sensors. Therefore estimating the pose of the robot only from these information will be inaccurate. Similarly, the data from sensors that scan the environment of the robot are also erroneous. Sonar based sensors are not very accurate for long distance sensing. Laser range finders give erroneous readings for reflective surfaces. Environmental factors like day light conditions, weather can distort the camera images. It is by means of statistical signal processing techniques like Kalman Filters, Histogram Filters or Particle Filter, that the localization algorithm recursively operates on a stream of noisy input data to produce a precise estimate of the position of the robot.

To summarize, it can be stated that localization is the means of deriving the position of a robot navigating in a known environment from noisy measurements made on the robot motion and robot environment. If the starting position of the robot is also known, the tracking of the position is possible by reading odometry information and using principles of dead reckoning. But due to the inaccuracy in the data received, the uncertainty of the robot position also increases with motion over time. In order to reduce this accruing uncertainty, the robot also has to retrieve its location by localizing itself with respect to the map, which has accurate information about this environment. For doing so, the robot makes observations of its environment using sensors and by a mathematical model relates the information obtained with the actual information from the map. Most common information used is the distance the robot senses to a landmark from its real world position, and the actual distance at which the landmark is on the map from the same position of the robot. The difference in position is the error in the measurement recorded by the sensors.

Probabilistic techniques for solving the localization problem applies the Bayesian Filtering method, which requires mathematical models of the behavior of the robot and the sensors These two models are known as the *Motion Model* and the *Measurement Model* and corresponds to the prediction step and the measurement update step of the Bayes Filter. The basic fundamentals of the models are described next.

Motion Model

The motion model describes the kinematic equations that govern the motion of the robot. In terms of probability, it is the state transition probability which describes the relationship of the pose of the robot at time step t with the pose at previous time step $(t-1)$ after the motion command u_t, by a conditional

probability distribution $p(x_t|x_{t-1},u_t)$. This is a posterior distribution and constitutes the prediction step of the Bayes Filter.

Two probabilistic motion models as described in [20] are the *Velocity Motion Model* and the *Odometry Motion Model*. The velocity motion model assumes that a robot is controlled through two velocities, a translational velocity (v_t) and rotational velocity (ω_t) at time *t*, and these values are passed on to the robot as commands to actuate the robot motors. The odometry motion model processes odometry information from proprioceptive sensors connected to the robot.

(**Definition**)*Proprioceptive sensors* – are internal or self-monitoring sensors that measure values internal to the robot, such as motor speed, heading. Some common proprioceptive sensors are Shaft Encoders, Compass, Inertial Navigation System (INS), and Global Positioning System (GPS) to name a few.

Both the probabilistic motion models take into account the inherent noise in the robotic system and explicitly models noise by some form of statistical distribution like normal/Gaussian distribution, triangular distribution, etc. Some common sources of motion noise are imperfect actuators, unsuitable terrain, or inaccurate sensors.

In this project, the MCL simulation is based on a modified version of the Velocity Motion Model for mobile robot navigation in planer environments. The noise is assumed to be Gaussian as a zero centered random variable with a given variance, obtained by experimentation.

Measurement Model

Measurement models describe the process of extracting measurement data z_t using the exteroceptive sensors in a particular robot pose x_t. The measurement model relates the measurement data with the state of the robot. These data provide information about the physical world around the robot.

(**Definition**) *Exteroceptive sensors* – are the external sensors that provide information about the robot's environment from the robot's frame of reference, such as distance to objects. These sensors are categorized as a proximity sensors. Some common examples of such sensors are laser range finders, sonar sensors, cameras in vision-based sensors and contact sensors like bumper sensors.

Probabilistic measurement models computes a conditional probability distribution $p(z_t|x_t,m)$ where x_t is the robot pose at time *t*, z_t is the measurement obtained at time *t* and *m* is an array or list of objects that represents the map of the environment. The inaccuracies of the sensors is the noise in the measured data and is represented explicitly by some form of statistical distribution usually Gaussian. A common measurement model used is that of a range finder that returns the distance to objects on its path. Sensors continuously generate values while in action. For the purpose of simulation and practicality, it is assumed

that a model of a range finder will return an array of K measurements z_t^k at time t.

$$z_t = \{z_t^1, z_t^2, z_t^3 .... z_t^K\}$$

where each element is an individual measurement.

In an ideal case, the probability $p(z_t | x_t, m)$ computed over these measurements is obtained as the product of the individual measurement likelihoods. This is used in particle filter methods to calculate the likelihood of the measurement, given as

$$p(z_t | x_t, m) = \prod_{k=1}^{K} p(z_t^k | x_t, m) \tag{2-10}$$

As mentioned in [20], robot localization problems can be classified into three different types, (1) Global localization, (2) Local position tracking, and (3) Kidnapped robot problem. In *Global localization* the initial position of the robot is unknown, and the aim is to correctly estimate the pose of the robot starting with the initial global uncertainty. In *Position tracking*, the pose of the robot is continuously tracked over time, as the robot moves. The initial position is often known and the uncertainty in robot pose is less since it is confined to a space around the robot's true pose. Local position tracking can be considered as the extension of the global localization problem, in which after the robot's pose has been correctly estimated and known, there is the need to continuously localize it so that the robot does not get lost. The third and the most complex problem in localization is the *Kidnapped robot problem* in which a localized robot can suddenly lose its correct pose tracking by moving it to an unknown space, or if the robot is not able to sense landmarks and objects in its map and assumes to be lost. Monte Carlo Localization can be used for accurate robot localization for all these problems[32].

Monte Carlo Localization

Monte Carlo Localization (**MCL**) [3], [32] refers to the use of particle filters and Monte Carlo methods[33] to solve the problem of robot localization as described above. It is a probabilistic method that derives itself from the Bayes Filter algorithm for state estimation. It calculates the belief of the states by Bayesian filtering methods conditioned on available measurements.

The principle idea of the MCL algorithm is to represent the probability density function or belief of the state of the robot by means of randomly drawn samples spread uniformly over the entire state space. These discrete set of samples are referred to as particles. The state variable is the robot pose, which changes as the robot interacts with its environment by a series of control actions, described by the motion model. The measurement model is then used to calculate the importance weight for each of the particles. At the end of every time step *t*, the pose of the robot is represented by a new sample of particles obtained from the resampling step. Keeping different values of the particles enables the MCL algorithm to maintain a multi-modal belief of the system. The MCL is a particle filter based algorithm and therefore is recursive. It continuously estimates or keeps track of the robot's pose from the available erroneous sensor data by applying the motion model, measurement model and the resampling step. The estimate of the robot pose can be obtained either as the weighted sum of all the particles, or the highest weighted particle can also be the best estimate of the robot pose.

The rest of the chapter describes the mathematical model for the Monte Carlo Localization using particle filters which executes in three steps: Prediction (Motion Model), Update (Measurement Model) and Resampling.

Prediction Model

The prediction model relates the motion model for mobile robots with the sampling step of a SIR particle filter, as discussed in earlier sections. The prediction model predicts the next pose of the robot based on the kinematics of robot motion. The noise in the robotic motion is explicitly modelled by an additive Gaussian noise model, a valid assumption for this research.

Figure 2-1 presents a graphical illustration of the kinematic model of robot motion and the effect of drift on the final position of the robot.

Assuming the robot's initial pose to be $[x_c, y_c, \theta_c]$,
the motion $[\Delta x, \Delta y]$ to
final pose $[x'_c,\ y'_c,\ \theta'_c]$
can be described in two steps :

a rotation,

$$\delta\theta = \theta_c^{'} - \theta_c$$

in which the robot first turns to face the next position, then followed by a translation,

$$\rho = \sqrt{\Delta x^2 + \Delta y^2}$$

to the next location.

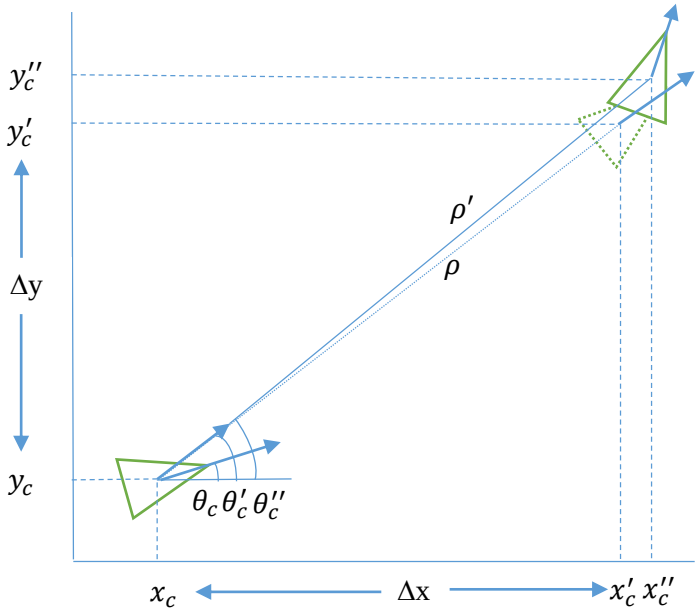


Figure 2-1 – Kinematics model of a mobile robot

This sets up the trigonometric equations

$$\Delta x = \rho\cos\theta_c^{'}$$
$$\Delta y = \rho\sin\theta_c^{'}$$

from which the final pose can be calculated as below

$$\begin{bmatrix} x_c^{'} \\ y_c^{'} \\ \theta_c^{'} \end{bmatrix} = \begin{bmatrix} x_c + \rho\cos(\theta_c^{'}) \\ x_c + \rho\cos(\theta_c^{'}) \\ \theta_c + \delta\theta \end{bmatrix} \tag{2-11}$$

The noise in the robot motion is applied independently to both the rotational and translational motion. Drift in the translational motion prohibits the robot's movement to be in a straight line and can be modelled by adding both rotational and translational noise to the forward motion.

Assuming a Gaussian distribution for noise with a mean μ proportional to the distance travelled by the robot in each time step and sigma σ which is a measure of the noise in the mechanical system, random noise samples are added to both the equations of motion, such that

$$\theta_c^{"} = \theta_c + \delta\theta + N_{rot}(\mu_{rot}, \sigma_{rot}^{\ 2}) \tag{2-12}$$

and

$$\rho^{"} = \rho + N_{trans}(\mu_{trans}, \sigma_{trs}^{\ 2}) \tag{2-13}$$

The pose of each particle in particle set X_t is sampled from the motion model, the input to the model is the previous pose at time *(t-1)* X_{t-1} and new particle set X_t is the output which represents the new probability distribution for the robot pose.

This corresponds to step 1 of the Bayes Filter, calculating the pdf $\boldsymbol{p(x_t \mid x_{t-1}^i, u_t)}$ (equation 2-7) for each particle i.

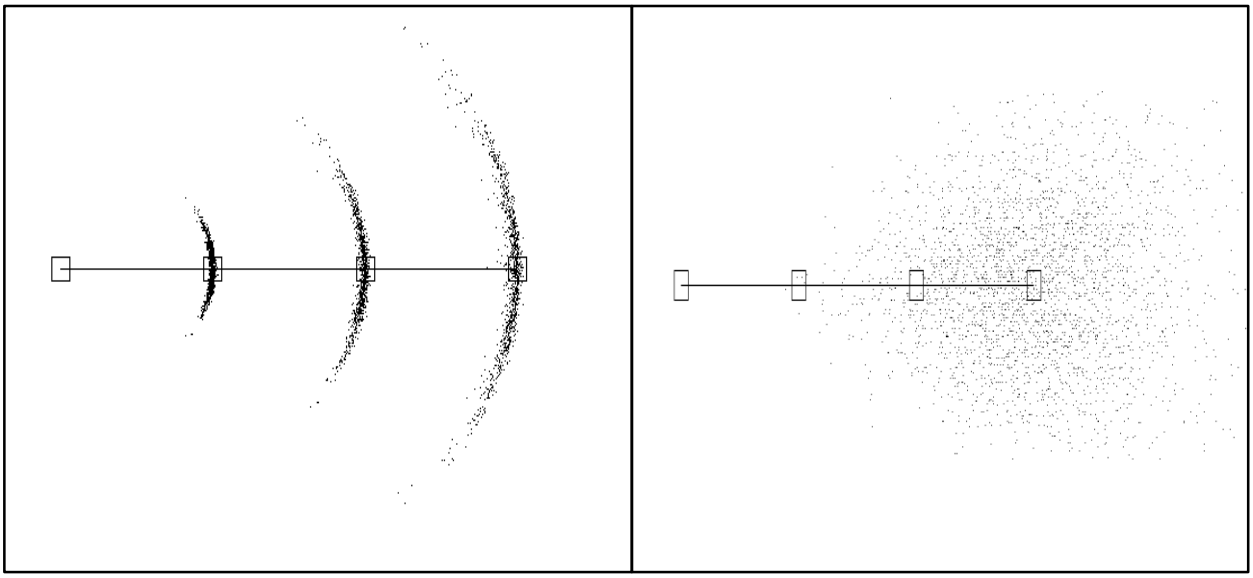


Figure 2-2 Uncertainty in motion model, high σ_{rot} ***Figure 2-3 Uncertainty in motion model, high σ_{trans}***

In Figure 2-2 the standard deviation of rotational noise is large, which introduces a larger uncertainty in the bearing orientation of the robot.

In Figure 2-3, high translational and drift noise result in the observed uncertainty around the robot's pose which is very high. The uncertainty is in both the position and orientation of the robot.

Update Model

After the robot moves ahead by one time step, the predicted pose of the robot is updated by the measurement model by incorporating the data from sensor measurements of the environment. The robot updates its position in two steps: - first by querying its sensors for measurement data and then by making a comparison of the sensor data with the true measurement from the map. The true measurement is with respect to the position of each particle in the map. An importance weight is assigned to each particle which is a measure of the quality of the particle in representing the true position of the robot in the real world. Higher the weight assigned to a particle, better is the accuracy of the hypothesis.

The weight of each predicted particle *i* is calculated using Equation 2.8 and Equation 2.10,

$$w_t^i \propto p(z_t \mid x_t^i, m) \propto \prod_{k=1}^{K} p(z_t^k \mid x_t^i, m) \tag{2-14}$$

where $p(z_t \mid x_t^i)$ is the likelihood function stating the probability of making the observation z_k from state x_k^i for *K* landmarks in the map.

Assuming a Gaussian noise model, the importance weight of each of the particles is then equal to the pdf of a normal distribution for state *x*, given by

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\{-\frac{1}{2}\frac{(\mu - \sigma_p)^2}{\sigma_p^2}\} \tag{2-15}$$

Substituting Equation 2-15 in Equation 2-14,

$$w_k^i = \prod_{k=1}^{K} p(z_t^k \mid x_t^i, m) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\{-\frac{1}{2}\frac{(d_{p_i k} - d_{rk})^2}{\sigma_p^2}\} \tag{2-16}$$

where d_{pik} is the distance of the i^{th} particle from the k^{th} landmark and d_{rk} is the sensor measurement data.

The Update model implements the second step of the Bayes Filter, by updating the value of the predicted state x_t from the measurement z_t using the inverse measurement probability, which is also the likelihood function, to get an estimated value for the particle samples representing the state x_t.

Resampling

As mentioned in the discussion of Particle Filters, the degeneracy problem suffered by the algorithm due to particle depletion can be mitigated in the resampling stage, where particles are re-sampled from the weighted set of particles in such manner that particles with higher weights are selected more frequently than particles with lower weights. This is repeated *N* times for *N* particles in the original sample set.

The resampling method used in this research is known as *stochastic universal sampling* method [20].This is also referred to as the *systematic resampling method*. The implementation of the algorithm is shown in Table 2.1

Input: X_t, W_t

1. $\overline{X_t} = \emptyset$
2. $index = rand(0, \mathrm{N})$
3. $c = w_t^{[1]}$
4. $i = 1$
5. $for\ n = 1\ to\ N\ do$
6. $\quad U = w_t^{[index+(n-1)]}$
7. $\quad while\ U > c$
8. $\quad\quad i = i + 1$
9. $\quad\quad c = c + w_t^{[i]}$
10. $\quad end$
11. $\quad add\ x_t^{[i]}\ to\ \overline{X_t}$
12. end

Output: $\overline{X_t}$

Table 2-1 Pseudo code for Low Variance Sampling

The input to the algorithm is the sample set X_t from the predict step, along with the measurement likelihood W_t computed for each particle in the update step. The output of the algorithm is the re-sampled set denoted by $\overline{X_t}$. The algorithm uses a single random integer between 0 and N (the total number of

particles) (line 2) to initialize the threshold value (line 5). The comparison begins with the weight of the first particle (line 3). The for loop (lines 4 – 11) iterates *N* times to resample *N* new particles. In the while loop (line 6 – 9), in every iteration a new sample is drawn based on the index *i*. If the weight for that particle is less than the threshold value U (line 6), the index is incremented by 1 to point to the next particle weight in the set W_t. The previous weights are also added to get the cumulative weight (line 8). The while loop terminates for the first particle at index [i] such that the corresponding sum of weights is greater than the threshold value. Then the particle[i] is sampled and added to the new particle set $\overline{X_t}$ (line 10). The for loop iterates by adding the weight of the next particle to the threshold (line 6).

The running time complexity for the universal sampling method is **O(N),** where *N* is the total number of particles. The output of the resampling step generates a new set of particles, which then becomes the prior belief for the prediction model in the next iteration of the MCL. At any point, the set of the particle samples represent the state of the robot.

Combining the mathematical models and equations for each step of the MCL, Table 2-2 describes the complete implementation of the MCL algorithm.

Input: X_{t-1}, u_t, z_t, m

1. $\overline{X_t} = X_t = \emptyset$
2. $for\ each\ particle\ p\ in\ x_t^{[p]} \in X_{t-1}$
3. sample $x_t^{[p]} \sim p(x_t|u_t, x_{t-1}^{[m]})$ **(Prediction)**
4. $w_t^{[p]} = p(z_t|x_t^{[m]})$ **(Update)**
5. $\overline{X_t} = \overline{X_t} \cup \{x_t^{[p]}, w_t^{[p]}\}$
6. end
7. $for\ p = 1\ to\ N$ **(Resampling)**
8. $select\ x_t^{[p]} \propto w_t^{[p]}$
9. $add\ x_t^{[i]}\ to\ X_t$
10. end

Output: X_t

Table 2-2 Pseudo code for Monte Carlo Localization

CHAPTER 3

PARALLEL IMPLEMENTATION

This chapter discusses the motivation behind the parallel implementation of the sequential Monte Carlo Localization algorithm described in Table 2-2.

Need for parallelization

Particle Filters compute state estimation by representing the probability distribution of the state variable by a set of random samples drawn from the posterior distribution (pdf) of the state. Such a representation allows the particle filter based solution to estimate even non-linear and non-gaussian systems [9], an advantage over other probabilistic based methods like Kalman Filters. Particle filters can also represent multi-modal beliefs, which is why they are able to globally localize a robot in MCL, by keeping multiple hypothesis on the robot's position. However, such sample based representations makes the particle filter an approximate method.

The robustness and accuracy of particle filter implementations, among many factors, directly depends on the number of particles used in the solution [32]. As discussed in the theory of Particle Filters and then extended to robot localization in the Monte Carlo Localization technique in Chapter 2, each particle or sample maintains a hypothesis of the robot position, that is, each particle represents the possibility of being the true estimate of the robot pose. Therefore, the more the number of particles, the better is the hypothesis and hence better is the accuracy in final estimation. But as the number of particles increases, the execution time of each step in the algorithm also increases, as each particle is independently sampled, weighted and re-sampled in order to provide the state estimate.

The execution time of a particle filter algorithm increases linearly with the number of particles is shown. The execution time is a measure of the high computational requirement which renders particle filter solutions ineffective for many real-time estimation problems. This also leads to a tradeoff between the desired accuracy and computation speed for particle filters.

In case of MCL, if the execution time for the algorithm is very high for a large number of particles, the frequency at which the robot can sense new data is limited which means the speed of the moving robot is restricted. As the execution time also depends on the number of landmarks observed at an instant, a large number of measurement data obtained from the sensors might make the algorithm computationally intractable. Therefore not all the data obtained can be used in localizing the robot. This results in loss of valuable information. Thus, to proceed with a successful implementation, care has to be taken to move the

robot at a velocity such that the MCL completes processing the incoming sensor data before a new stream of data arrives. Also, the number of particles have to be so chosen that the robot does not lose track of its position or fail to globally localize itself. These limitations have been addressed to a certain extent in [3]. In this paper, the authors implement a strategy in which the number of particles used varies with time.

Performance improvement of algorithms by means of parallelizing them to take advantage of multi-core processors has always been of high interest to research scientists. A parallelized implementation of the MCL can improve the computational performance and computational efficiency without affecting the accuracy. This is possible by re-implementing the MCL such that the algorithm can take advantage of all the multiple CPUs in a multi-core processor and execute in a truly parallel fashion to reduce the total execution time by distributing the work among the CPUs.

Parallel Particle Filters

The flow of work in different stages of a particle filter algorithm like MCL, described in Table 2-2 can be explained by the flowchart in Figure 3-1. The flowchart clearly suggests an implicit data parallelism [34] in the particle filters which can be taken advantage of by a suitable mapping of work [35] to different CPUs in a multi-core processor.

The recursive steps of the algorithm consists of the processes: (i) State prediction (sampling), (ii) State update (likelihood calculation), and (iii) Resampling, which are implemented sequentially in one simulation step. However, as seen the flowchart, the first two processes – Sampling and Likelihood calculation can be applied to each individual particle independently. The resampling step implements the *Low Variance Sampling* algorithm which is executed in a sequential fashion, as selection of each particle depends on its weight as well as the weights of its neighboring particles. The first two processes exhibit data level parallelism in the algorithm, in which the same operations – prediction of next pose, and calculation of importance weight are applied concurrently on each particle. This data parallelism has been exploited in this research to achieve the maximum parallel speed up in the algorithm.

The steps in the algorithm are summarized as follows:

Initialization - The initial position of the robot is unknown (assuming global localization) and therefore all the particles in the set are sampled randomly from a uniform distribution. The only operation in this step is to assign random numbers to the particles such that they cover the entire state space and the particles are distributed across the entire map. Initialization is done only once at the beginning of the simulation. As this step is not a part of the recursive MCL, it has not been implemented in parallel in this research. However, parallelization of this step is possible.

Sampling - Each particle is sampled from the previous particle set by applying the kinematic equations defined in Chapter 2 (2-11). Random noise from a Gaussian distribution is added to the sampled pose (2-12, 2-13). This step can be trivially parallelized for each particle, as each particle x_t^i is predicted from its previous value x_{t-1}^i and control data u_t and does not depend on the states of other particles.

This parallel processing is denoted MCL in Figure 3-1, by the first join/fork symbol which shows the forking of each particle into separate and independent paths from the Sampling till the Likelihood calculation step of MCL.

Likelihood Calculation – The importance weight for each particle is calculated by from the Normal distribution (2-15) function (assuming Gaussian noise). This is also a parallelizable step in the algorithm as the importance weight of each particle only depends on the number of landmarks or measurement data observed by the sensor (2-16).

Resampling – The resampling step shown in Table 2.1 has data dependencies and cannot be parallelized. The data dependency exists because the algorithm chooses particles based on their weight and each particle is processed sequentially to compare their weights with the neighboring particles till a certain threshold is reached.

The second join/fork symbol in Figure 3-1 shows the join operation in MCL before the resampling step. This signifies that resampling can begin only after all the particles have been sampled and have an importance weight calculated for their predicted value.

State Estimation – The state estimation is an optional step to estimate the pose of the robot at the end of each cycle. There are different techniques to get the state estimate. The two commonly used are *weighted sum of particles or the highest weighted particle*. In this research, the highest weighted particle is chosen to be the best estimate of the robot pose. As the computation in this step is relatively basic and non-parallelizable, the execution time for state estimation is considered along with the resampling step and together forms the total sequential time of execution for the algorithm.

The estimated pose can be used to calculate the accuracy of the algorithm in simulation or if the robot pose is needed for further processing, for example in Simultaneous Localization and Mapping(SLAM) [4][36], the pose of the localized robot is needed to build a map of the environment of the robot.

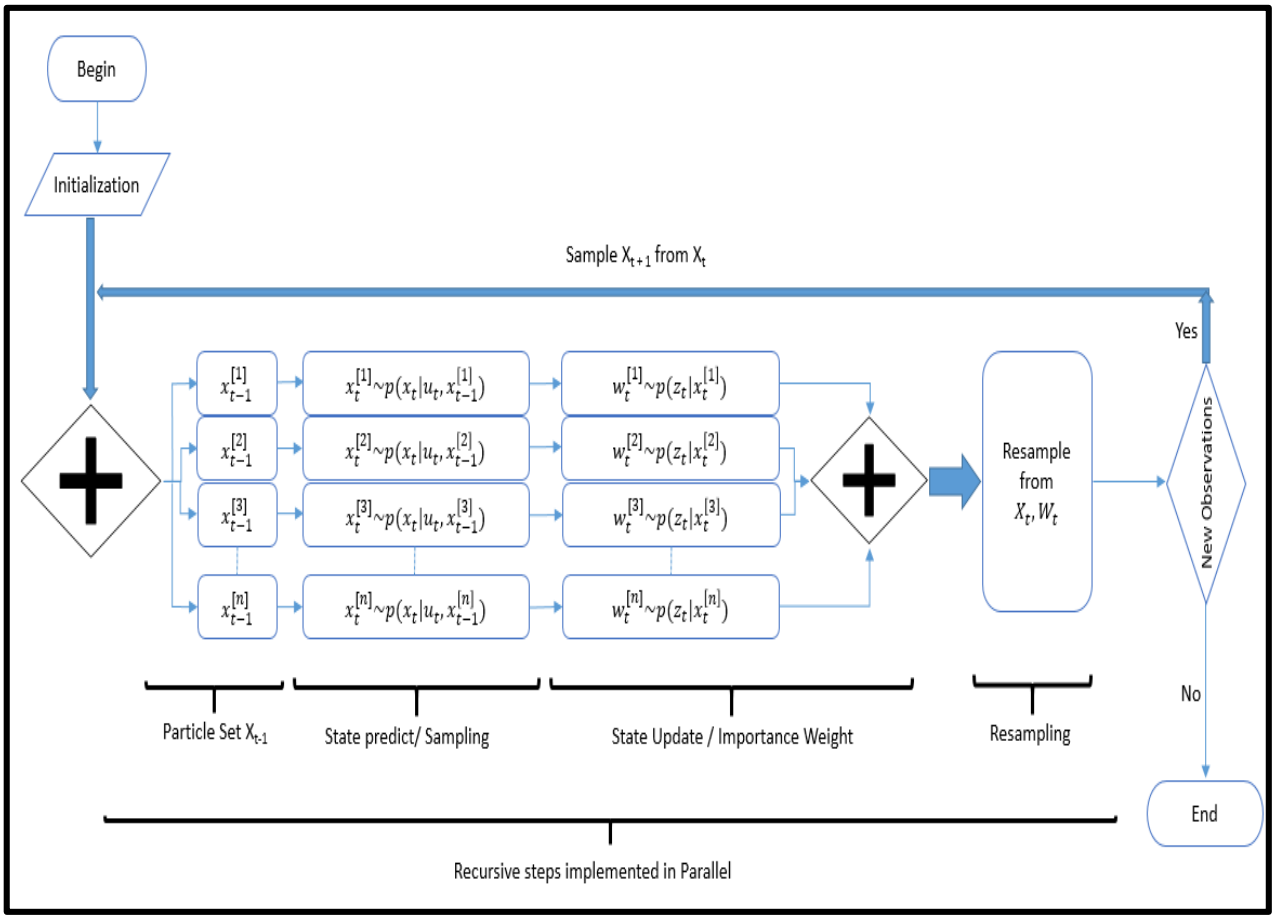


Figure 3-1 Work flow in a Particle Filter / MCL implementation

Chapter Four describes the simulation model for robot motion, Chapter Five describes the experiments on the parallel and sequential implementation of the MCL, and Chapter Six shows the results of the experiments, viz. execution time and speed up from the parallelized version, compared against the sequential version.

CHAPTER 4

SIMULATION

This chapter describes the simulation model developed to implement the Monte Carlo Localization algorithm. A visual demonstration of the working of the MCL for global localization and local position tracking is also shown.

In order to implement the MCL algorithm for mobile robot localization in a known environment, a map of the environment. The map is the collection of the objects in the environment along with some known properties such as an object's location. For this project, it is assumed that a planar map is provided to the MCL with the Cartesian coordinates of the landmarks and each landmark being uniquely identified by an index number. Providing the index number of the landmarks allows the use of a sensor model with known correspondence [20].

The second requirement is to have a mobile robot system that continuously generates sensor data and control data. In this research, the robotic behavior is a computer simulation model that simulates a mobile robot and a range finder that generates the measurement data which are distances from robot to landmarks, in the Cartesian coordinate system. The control data simulates action commands to the robot to either move ahead or steer clockwise or anticlockwise. The control actions are used with an odometry motion model for MCL simulation. The details of the simulation model are discussed below.

Robot Simulation Model

The simulation model has been developed in C# and Visual Studio. The map is generated as a *location-based map* with a set of points as landmarks with Cartesian locations and a unique index number for each point.

The `CreateMap()` method generates the map, an array of landmarks, with the number of landmarks being passed as an argument, 48000 landmarks have been used to test the simulation. The map represents a corridor like structure as shown by the two blue lines in Figure 4-1. This array of map is the input '*m*' as described in Table 2-2, and is passed as an argument to the MCL simulation.

The movement of a robot is a continuous action, which is difficult to analytically model in a computer. Therefore, the robot movement is modelled by a number of discrete steps, 200 steps in one simulation, to be able to describe the effect of noise in the movement of a real robot without making the whole simulation computationally intensive, which would otherwise affect the timing characteristics of the MCL also being performed in the same computer. For this research, it is assumed that in each discrete step,

the robot can move ahead by a fixed distance of 0.5 units, or turn clockwise or anti-clockwise by 60 degrees while having a small translational movement of 0.1. The translational movement during the steering action is very important to bring the discrete simulation close to continuous behavior during robot steering as it is impossible that a real robot can suddenly change directions by 90 degrees in its continuous motion.

The `RobotMotionCommandGenerator()` method randomly generates a set of control data (u_t) as command actions for forward and rotational movement of the robot. The data is generated in terms of 0, 1, and -1 for the purpose of reducing floating point computations, and also serves the purpose of the simulation. A command object will contain two values, `goForward` and `steering`; `goForward` can be either a 0 or 1 and *steering* can take values 0, 1 or -1. This is because in one time step, the robot can either move forward or steer anticlockwise or clockwise. Thus, the control data generated can be one of the possible values,

(1,0) – robot moves forward,

(0,1) – robot steers anticlockwise,

(0,-1) – robot steers anti clockwise.

Here it is assumed that anticlockwise is the positive direction of rotation and clockwise negative.

The control data is stored in a list, and for every step of the simulation a new data is read according to which the robot moves to a new position. The robot action is implemented in the `MoveRobot`() method which receives the previous position of the robot and values of the *steering* and *goForward* variables as arguments. It computes the next position of the robot based on the kinematics equations (2-11) in Chapter 2. The command data is also processed by the motion model implemented by the method `SampleMotionModel()` which also takes as arguments the values of *steering* and *goForward,* and the previous known state of the particle. Noise is added to the predicted values of the particles according to (2-12) and (2-13).

The measurement data z_t is generated by the `SenseLandmarks()` method. The method returns an array of measurements that includes the distance of the landmark and the angle between the landmark and the robot pose. The maximum field of view is 180 degrees, ±90 degrees from the pose of the robot, and the maximum range is 3 units. The second assumption made is the maximum number of valid measurements the sensor will return which is capped at 181- one range measurement per degree. This also avoids dynamic memory allocation in the simulation.

The working of the MCL simulation is described in Figure 4-1 and Figure 4-2.

Verification of the Simulation Model

Global Localization

As discussed in Chapter 2, **in Global Localization**, the initial position of the robot is unknown, except that the robot is within the map. Therefore in the first step, the particles will be initialized over the entire surface of the map, where each particle represents a hypothesis of the robot's position.

Because of the use of the known data association model where each landmark has been identified with a tag number, the importance weight of each particle is calculated only against the observed landmarks. This leads to very quick convergence of the particle's pose to the actual robot pose. This is one advantage of using a location based map as compared to a feature based map.

The correctness of the simulation model for the MCL has been verified by checking the convergence of the particles towards the robot position. This experiment is a proof that the particle filter based MCL simulation model used in this research to examine parallelism accurately estimates the position of the navigating robot.

Local localization

Also discussed in Chapter 2 is the problem of **local position tracking,** which is to keep track of the robot's pose once it has been successfully localized in the map. At every simulation step, the sensor module returns an array of distance to landmarks and landmark IDs. Again, due to the known data association model, particles with more accurate hypothesis on the robot's position gets a much higher weight in comparison to others and thus get reselected more often based on their relative weights.

Figure 4.1: illustrates the steps in the robot simulator for Global localization.

Figure 4.2: illustrates local position tracking in the robot simulator.

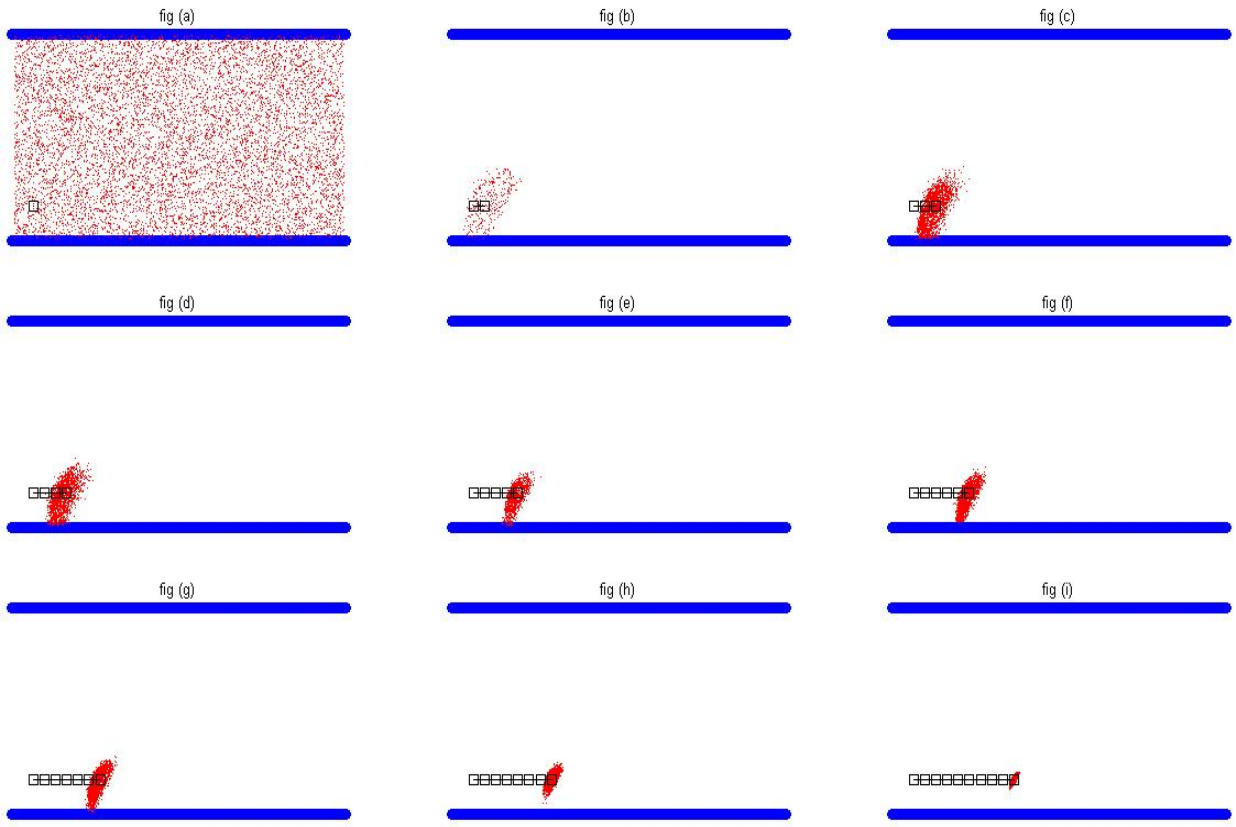


Figure 4-1 Global localization in the simulation model

Description:

Figure 4-1 shows various stages of the MCL algorithm during global localization, as the robot, shown by the black square, navigates in a straight line and senses its environment after each time step. The small discrete steps can accurately represent continuous motion of the robot.

Figure 4-1 – Fig. (a) – shows the initialization of the particles over the entire map. The starting position of the robot is observed (black square). The position of the robot is unknown to the MCL, but shown here to verify that the particles converge to the correct position.

Figure 4-1 – Fig. (b) – The robot starts moving. At the end of first time step of MCL, the cloud of particles have converged around the robot but with a higher uncertainty.

Figure 4-1 – Fig. (c) - This shows the position of the robot and cloud at the end of second simulation step. The density of the particle cloud has increased around the robot as compared to Fig b.

Figure 4-1 – Fig. (d) to (h) - With each subsequent time step, more and more particles converge towards the exact position of the mobile robot. The particles with higher weights are replacing the particles with lower weights.

Figure 4-1 – Fig. (i) - In figure i, it can be claimed that the MCL has accurately estimated the global position of the robot with respect to the map, thus verifying the simulation model.

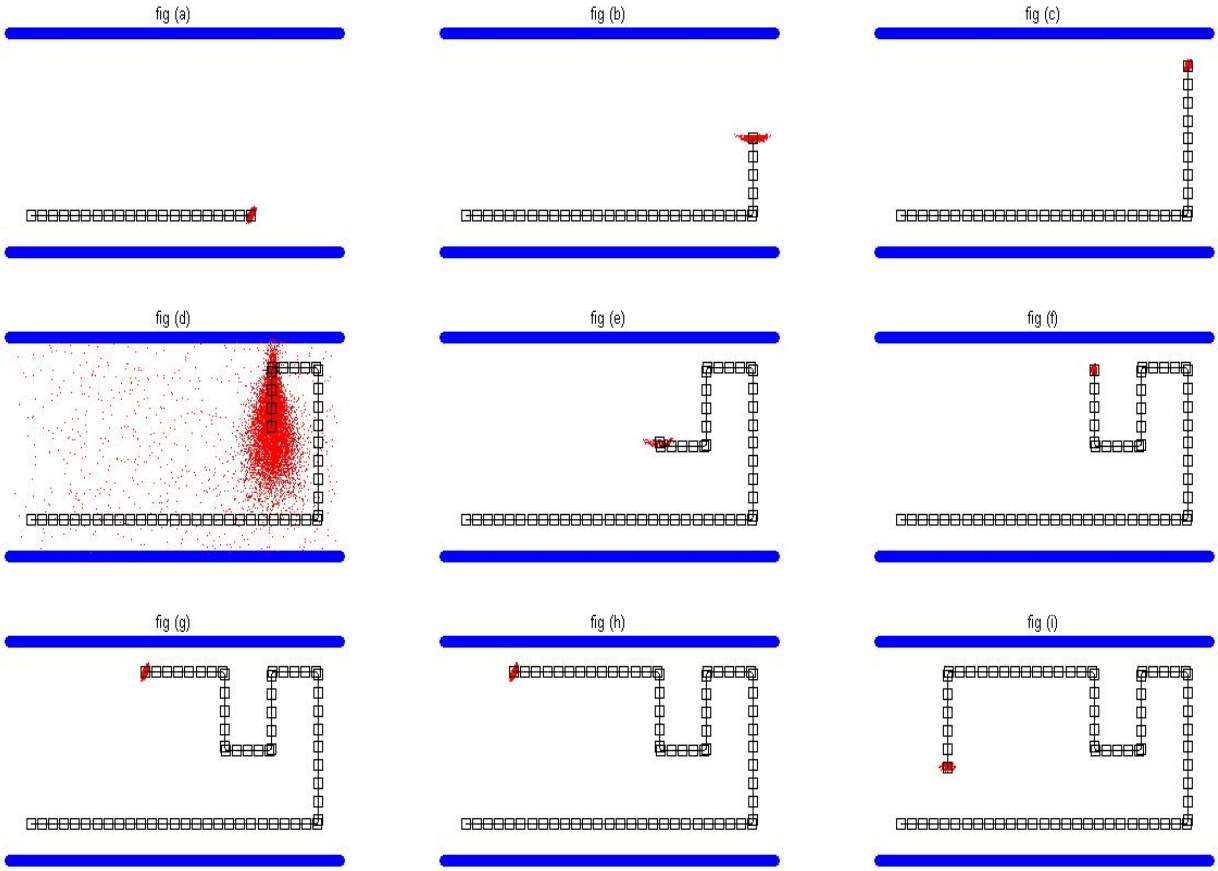


Figure 4-2 Continuous Position tracking in the simulation model

Description

Figure 4-2 shows various stages of the MCL algorithm after the estimation of the global position of the robot. The robot continues navigating along the corridor, and it can be seen that the cloud of particles accurately track the robot position.

Figure 4-2 - Fig (a) – the particles correctly estimate and track the robot position.

Figure 4-2 - Fig (b) – the particle cloud is seen to diverge in this figure, this happens because of the robot position in the center of the corridor where it observes very few landmarks. This adds uncertainty in state estimation.

Figure 4-2 - Fig (c) – as the robot reaches near the wall, the particles converge again with much higher certainty.

Figure 4-2 - Fig (d) – the direction of movement of the robot makes the sensor fail to read any landmark thus adding to high uncertainty in robot position. Random particles are generated across the map in order to make sure MCL is able to re-estimate the robot position.

Figure 4-2 - Fig (e) to (h) – the robot moves and when it observes new landmarks, the MCL is again able to estimate the pose.

This demonstrates the correct working of the simulation model. It visually describes the recursive nature of the Monte Carlo Localization and the importance of resampling to get rid of bad estimators or particles by replacing them with better particles.

In the subsequent sections, the execution time for different configurations of the MCL is discussed. Here, the focus would be on the computational efficiency of the algorithm rather that accuracy, which has already been shown to very high due to certain assumptions and modelling of the simulation.

CHAPTER 5

EXPERIMENTS

The following section describes the experiments performed to analyze the sequential and the parallel implementation of the MCL algorithm. The results are discussed in Chapter 5.

Test Environment and System Configuration:

Hardware

The performance of both the sequential and parallel implementations of the MCL has been tested on an *Intel Core i7-2630QM* quad core processor capable of running 8 threads concurrently, most of the results have also been verified on an *Intel Core i5-2410M* which is a dual core processor in which 4 threads can be run in parallel. The architectures of both the processors are same, while the memory configuration is significantly different. In the remaining of the thesis, they will be referred to as i7 and i5. The complete processor configuration is mentioned in the appendix.

Software

The MCL simulation has also been implemented in C#, the parallel implementation has been developed using the Microsoft DotNet framework libraries for multi-threaded programming. The parallelization [37]–[39] has been achieved by using multiple number of threads to leverage the computational resources available in the above mentioned hardware.

Previous work on parallelizing a generic particle filter [40] uses *general purpose graphical processing units* (GPGPUs) to implement a parallel version, in [41] the OpenMP and Compute Unified Device Architecture (CUDA) programming libraries were used to parallelize a PF algorithm for video tracking.

The DotNet Framework provides only a uniform random number generator. The Box-Muller transformation algorithm[42] was used to transform uniform distribution samples to Gaussian distribution samples for adding noise in the system. The random numbers have also been made thread safe. Each thread had an independent random number generator which were initialized with a unique seeding value, so that different random numbers were generated for different threads when processed concurrently.

For the purpose of testing, each uniform random number used in the implementation has been seeded with unique values to regenerate the same sequence of random numbers in order to recreate the simulation repeatedly for different values of particle number and thread configurations.

Experiments

Experiment 1- Sequential Implementation

Experiment 1.1 Performance comparison of Sequential Implementation.

The first experiment was to compare the performance of the sequential MCL implementation on two different processors. The MCL was executed on the i7 and i5 processors to get the total running time as indicators of performance. The results have been plotted in Figure 6-1. The similar execution profile was a proof of the fact that a sequentially implemented code cannot take advantage of extra cores available in a processor. Thus, even though the state update and state predict methods in the MCL exhibit data parallelism as seen in Figure 3-1, there was no noticeable difference in performance from the two different processors. The second part of the experiment was to analyze the relationship of the execution time with the number of particles. Figure 6-2 shows the plot for the same for both the processors which was exactly the same. The execution time has a linear relationship with the number of particles.

Both the results served as the motivation to re-implement the MCL to achieve two goals. First, to make the code run in parallel such that it executed faster in a processor with more number of CPUs than on a processor with lesser or a single CPU. Second was to reduce the total execution time for the MCL for large particle sets.

Experiment 1.2 Performance analysis of Sequential Implementation

In this experiment, the different steps in the MCL were profiled based on their execution time in the sequential implementation. The sequential implementation only has a single thread which forms the base case. The results are plotted in Figure 6-3 and Figure 6-4.

It was calculated that more than 97% of the total execution time was taken by the state update step in which the importance weight of each particle with respect to all visible landmarks is computed.

The second most time-consuming step was the sampling process. Although for larger particle sets, it was noticed that the sampling process had execution time similar to the sequential portion of the code which included the resampling step and the state estimation step. The execution time of each step increased linearly with the number of particles.

Experiment 2- Parallel Implementation

Experiment 2.1 Parallel Implementation of the MCL

From results obtained in previous sections, the goal was to introduce explicit parallelism in the state prediction and state update method which were the most time consuming steps in the MCL. With parallelism, each particle in the code can run in parallel. This was achieved through multi-threading.

As shown in Figure 3-1, each particle in the MCL is first sampled through the sampling or state prediction process followed by the importance weight calculation in the state update method. This process is carried out for each particle in the sample set and there is no data dependency among the particles.

In order to parallelize the code as shown in the flowchart of Figure 3.1, each particle in the set had to be assigned to one thread, which were then set to be executed in parallel. Thus, this required initializing *N* threads for *N* particles. The method for the predict step and update step was passed as a single sequential process to each of the thread as a lambda expression [43].

The algorithm however synchronizes before the resampling step, so that before resampling begins, all the particle must be sampled from state X_{t-1} to a temporary state $\overline{X}_t$, and for each particle $x_t^{[i]}$ there must be a weight $w_t^{[i]}$ in the array W_t.

This was achieved by using the ***CountdownEvent*** class in the ***System.Threading*** namespace of the DotNet framework4.0.

However, running this code did not produce an expected result, rather the experiment was aborted as it continued beyond 3.5 minutes for only 512 particles; this was 150 times more than the sequential running time.

Experiment 2.2 Re-implementation using Parallel Class library.

Executing a large number of logical threads, much greater than the number of physical cores in the processor can cause oversubscription which results in the significant preemption in all the active threads. This adds considerable overhead.

In Experiment 2.2 the same idea was re-implemented using the Parallel Class (Task Parallel Library) in the .Net framework4.0. Instead of having *N* threads for *N* particles, the maximum number of threads that could run in parallel was set to the maximum number of logical cores in the system. This was obtained by querying the `Environment.ProcessorCount` variable.

Now, 8 threads for the i7 and 4 threads for the i5 can run in parallel, processing 8 and 4 particles concurrently for i7 and i5 respectively, till all the particles have been sampled and importance weight for each particle calculated. The result of the experiment is plotted in Figure 6-5.

The results from Experiment 2.2 describes hardware issues in applying multithreading. This relates to the cache memory size in the processor. The detailed analysis is presented in the results section.

The remaining experiments implements parallelization in the MCL by manually creating multiple threads and assigning tasks to each thread, such that each thread can execute the tasks in parallel.

The most important step in these experiments, which also distinguishes this research from other similar works, is that each individual thread created is also assigned a CPU in the multi-core, in which the thread is executed. As stated in [43], a thread in a process can switch over from one CPU to another during its execution, however each jump in the thread execution results in reloading the processor cache which affects performance. In order to get the best performance from each CPU it must be utilized up to 100% until thread exit, for all non-blocking threads. This can be achieved by specifying the CPU that should run a specific thread which reduces or completely avoids cache reloading. This association between a processor and a thread is called the processor affinity.

In C#, processor affinity can be set by the `Process.ProcessorAffinity` property. Each CPU in the multi-core processor is identified by a bit. For the i7 processor, running a Windows 7 operating systems, there are 8 logical CPUs. Therefore an 8 bit mask can be used to represent each one of the CPUs.

For example:

CPU 0 can be selected by the binary value (mask) - 00000001 = 01H

CPU 2 and 3 can be selected by the bit mask - 00000110 = 06H

Similarly, all the 8 CPUs can be selected by - 11111111 = FFH

Allotting a specific CPU or CPUs to user-created threads is not possible directly with the current application programming interfaces (APIs) available. It should be noted that the *ProcessorAffinity* property in a member of the class Process, whereas threads created by the programmer belongs to the class Thread.

In this research, the method proposed to assign CPUs to threads is by reading the current process in execution. All threads belongs to some process, therefore querying the process for all its current threads is an indirect way of accessing the user-defined threads. The code implementing this is described in the next page.

```
Process currentProcess = Process.GetCurrentProcess();
ProcessThreadCollection processThreads = currentProcess.Threads;
int totalThreads = processThreads.Count;
```

The problem with this approach is that the total number of threads in a process consists of both user-created as well as operating system created threads, plus the *Main* thread of every C# application. Therefore the threads in the *processThreads* collection have to be sorted by some means to select only the user-created threads.

It was observed that each thread explicitly created by the programmer is added to the end of the processThreads collection. Therefore, the last *N* threads in the collection are the programmer created threads. This implementation is shown below.

```
int z = 1;
for (int c = 0; c < numProcs; c++)
{
      int index = totalThreads - numProcs + c;
      processThreads[index].ProcessorAffinity  = (IntPtr)z;
}
```

In the above code, the value stored in variable *z* is assigned to the threads. For instance, in the above case, all the threads have been assigned the CPU 0.

As noted from the result of Experiment 1.2, the running time of the likelihood calculation step is about 97% of the total MCL, and the sampling step executes for 2% of the total time. Therefore in order to parallelize these sections, threads have been created for each of the sections. The amount of work assigned to each thread is determined statically by dividing the total number of particles with the number of threads. For an i7 the maximum thread count is 8, while for i5 is 4. This is known as static partitioning of workloads[37].

Such a load-balancing strategy works well here because the operations on each particle is the same. Therefore assigning an equal set of particles, and hence an equal amount of task to each thread makes load balancing perfect.

Secondly, both the steps of the algorithm are separately parallelized, unlike in the previous implementation. This is done to ensure better data localization in the cache memory and also true data parallelism, in which the same operation is applied on the data set.

The drawback of this approach is that the entire set of particles have to be iterated over twice, once for sampling and then for likelihood calculation.

Experiment 2.3 Parallelization of the Motion model / State Predict step.

As described previously, the parallelization of the motion model is performed by assigning a fixed and equal number of particles to each thread. Each thread is allotted a different CPU in which to run, to ensure minimum cache reloading due to thread switching. For best performance, for 4 or less number of threads, the CPUs assigned are either all even-numbered (CPU0, CPU2, CPU4, CPU6) or all odd numbered (CPU1, CPU3, CPU5, CPU7). The operating system starts indexing the CPU from 0. This assignment ensures that each logical thread is mapped into a different physical CPU. The synchronization among threads is done by using a CountdownEvent timer, initialized to the total number of threads used. As each thread finishes its task, the timer decrements by one. The countdown timer implemented in Experiment 2.2 was initialized to the total number of particles, as it was counting the execution of each particle, thus adding extra overhead for synchronization.

The motion model is computationally a simple step with the most time consuming operation being the random number generation. In fact, this step is more memory bound that CPU intensive, as it requires to read each particle coordinates to generate/sample new coordinates by applying the kinematics equation, which are simple mathematical operations.

The maximum speedup gained was **3.4** over the sequential version of motion model. The results are analyzed in greater detail in Chapter 6.

Experiment 2.4 Parallelization of the Measurement Model/ State Update step.

The parallelization of the Measurement Model is also done in exactly the same way as that in the motion model. The measurement model is a computationally intensive process as it calculates the weight for each particle from a normal distribution function (2-14). The weight is calculated w.r.t each landmark observed (2-15) and thus the final weight is a product of all individual weights of that particle for each landmark. The measurement model therefore has a second inner for loop for each particle and can be parallelized. However, parallelizing this loop did not produce good results because in order to obtain the product of all the individual weights computed in parallel, a lock is required to synchronize the product variable for concurrent access by each thread. This reduces maximum speed up as locks have a high overhead. Secondly, parallelizing both the inner and outer loop also does not produce the best performance because due to the static load partitioning of the particles among all the available threads, each thread is already busy with its own portion of work. The best performance was obtained by executing this for loop

sequentially within the same thread to which the particle was allocated. This produced the maximum speedup with minimum overheads resulting from thread synchronization or communication.

Experiment 2.5 Parallelization of the MCL

In this experiment the total execution time of the new parallelized MCL implementation is calculated. The system clock computes the total time for the MCL over the entire simulation of 94 steps. The average time is calculated.

The total time for the sequential part of the code that includes the resampling step is also calculated. The sequential operations in a parallelized code affects the speed up gain by parallelizing. This relationship is expressed by Amdahl's law [19][44].

The execution time obtained from the MCL is used in Amdahl's law to compute the theoretical speedup and compared against experimental speed up achieved.

CHAPTER 6

RESULTS

In this chapter, all the results for the experiments described in Chapter 5 is presented. The performance of the parallel implementation of the algorithm in terms of total execution time and speedup against the sequential single-threaded implementation is analyzed and verified. The results show a significant speedup improvement as the number of cores assigned for the parallel implementation increases. More importantly, the results also verifies that a trivial implementation in parallelizing the MCL does not produce the best result. This is applicable to other SIS based particle filter algorithm solutions. The parallel performance of the algorithm is directly influenced by the granularity of the tasks each thread performs under parallel execution. This is verified from the results as the speedup gained for larger particle sets is compared to smaller particle sets. The maximum speedup obtained was also verified from Amdahl's law. Amdahl's law give a theoretical upper bound on the speedup of an application, which was verified to be true for most configurations of particle sets and the number of threads used for parallelization.

However, it is also important to mention here that this simulation has been performed under several assumptions as discussed in Chapter 5. There is always a certain amount of discrepancy when modelling a continuous time event by discrete simulations. This applies to the simulation of the robot and the sensors only. As the robotic system has been simulated in the same machine running the MCL algorithm, hardware limitations on processor and memory usage can also affect the performance of the MCL under study.

The aim of the research was to develop an efficient parallel framework for particle filter based algorithms that will improve computational performance of such algorithms without any tradeoff with their accuracy. Here, efficiency implies to the execution time. The experiments and corresponding results on the MCL verify the parallelized implementation.

Sequential Implementation

Experiment 1.1 Performance comparison of Sequential Implementation.

The performance of a sequential MCL algorithm tested on the i7 and i5 processor is plotted in Figure 6-1. The processor specifications are listed in the appendix.

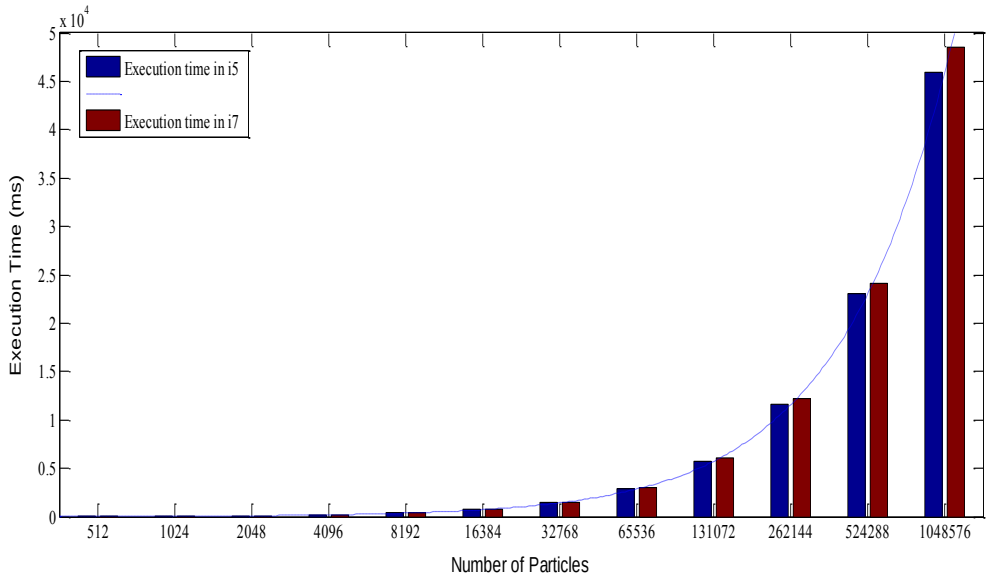


Figure 6-1 Execution time of sequential MCL on i7 vs i5

Figure 6-1 compares the execution time (in milliseconds) on the i7 and i5 processors as the number of particles in the algorithm increases. As observed in the plot, the sequential implementation of the MCL takes approximately the same time on both the processors which are significantly different in hardware configuration. Infact the code performs slightly better in the i5 than in i7 because of the higher clock speed of i5 CPUs. Though the i7 is a quad core processor with double the cache memory than in the i5, it is evident that performance of a sequential MCL or any similar particle filter application cannot make use of the extra computational resources. The only factor affecting the execution time is the clock frequency of individual CPUs.

The semi-log plot of the same data is shown in Figure 6-2

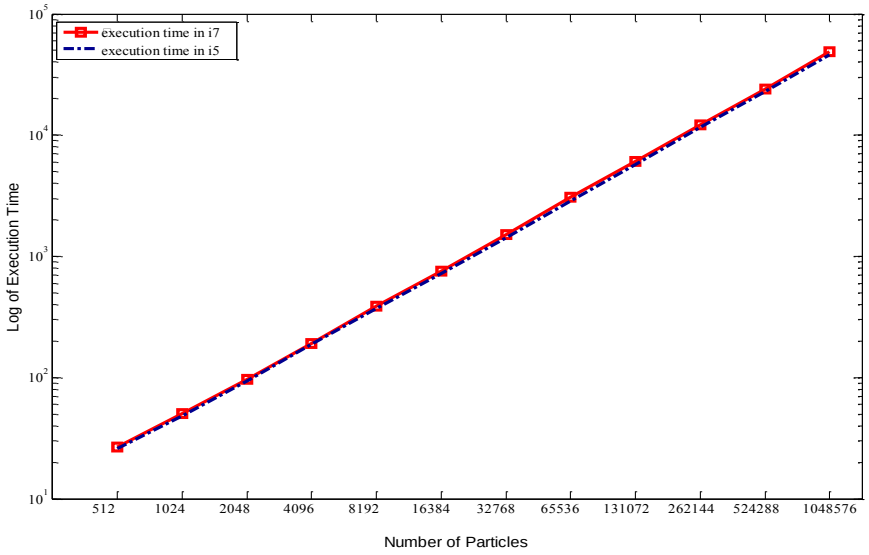


Figure 6-2 Execution time v/s number of particles

The overlapping lines in the plot of Figure 6-2 verifies the similar performance of the two processors for the sequential MCL.

The second inference that can be drawn from the above result is the increase in execution time of the algorithm as size of the particle set increases. In the log scale, it is a linear growth as the number of particles increase as the *power of 2*. This accentuates the underlying necessity for parallelizing the algorithm as the only means to improve the performance without changing the math or trading off with accuracy.

In Experiment 1.2, the execution time of the sequential algorithm is evaluated in detail so as to identify the most time consuming steps of the algorithm, which were then re-implemented in parallel.

Experiment 1.2 Performance analysis of Sequential Implementation.

The execution time of the motion model (sampling), measurement model (importance weight calculation) and resampling are shown in Figure 6-3 and Figure 6-4. The purpose was to identify the most time consuming steps in the algorithm among the three recursive steps of the MCL.

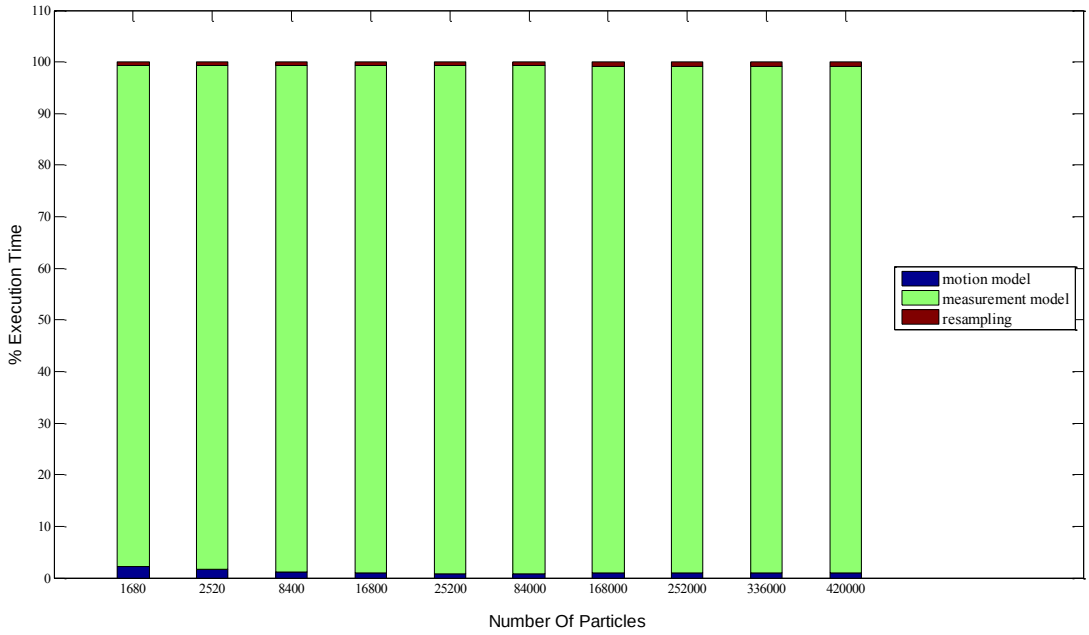


Figure 6-3 Percentage of execution time in timing profile of sequential MCL

In Figure 6-3, the percentage of the total execution time for each step is plotted against the number of particles. The height of each section in the bar graph represents the percent of total time for that step as identified in the legend of the plot. It can be observed that for each particle set the maximum time is taken by the measurement model. This is about 97% of the total time for the MCL. The time taken by the resampling step plotted in the figure also includes the state estimation time. It is the total time for the non-parallelized section of the code

In Figure 6-4, the log of execution time vs number of particles is plotted for the same data. This graph in the semi-log scale clearly shows the division of time among the three steps. Similar to the results in Figure 6-1 and Figure 6-2, the time for each step and the combined total time increases linearly with the number of particles.

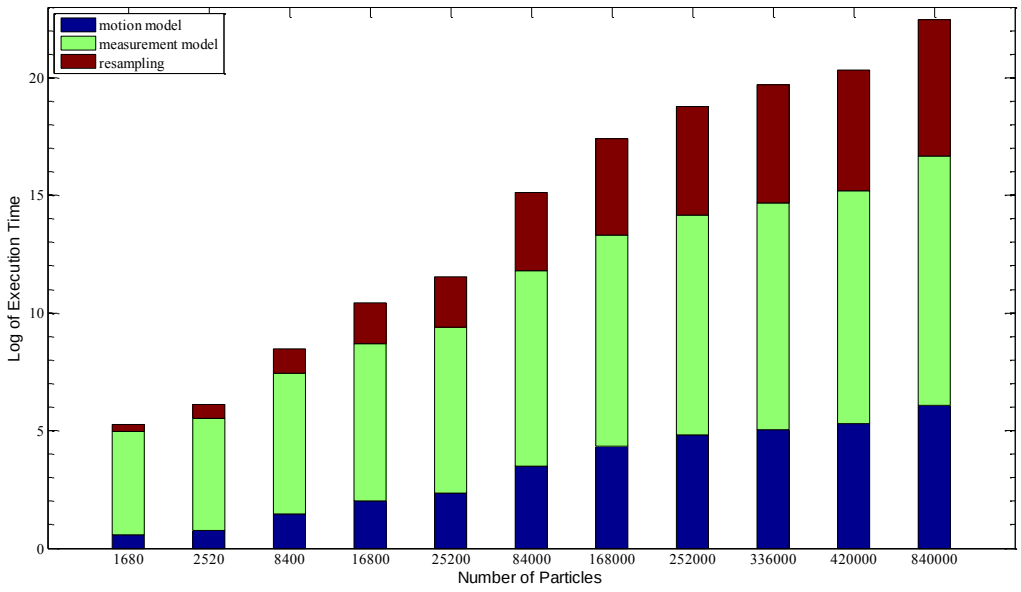


Figure 6-4 Log of execution time in timing profile of sequential MCL

The results of this evaluation leads to the conclusion that the measurement model/ update step that computes the importance weight for the particles is the most time consuming step of the algorithm. The motion model / predict step is the second most time consuming step followed by the resampling step. As already discussed, the resampling step cannot be trivially parallelized without affecting the accuracy of the algorithm. Thus, the motion model and the measurement model implementations were targeted to be re-implemented to execute them in parallel.

In the next section, the results of the parallel implementation are analyzed.

Parallel Implementation

In this section, the performance of the parallel implementation of the particle filter based Monte Carlo Localization algorithm is studied. The parallel implementation has been achieved by using multiple threads of execution [37]–[39] which allows to extract maximum performance from the underlying hardware. This is known as multi-threading. In this project, the Microsoft DotNet framework[14] has been used for all the parallel implementations.

The results obtained from the experiments conducted in this section are very important in understanding the performance of a parallel algorithm, in this case the Monte Carlo Localization algorithm, or a generic particle filter application. The experiments and the results demonstrate some of the intricacies in parallel programming. Also identified are the hardware and software limitations on maximum speed up and performance gain that can be achieved from multi-core processors. Hardware limitations includes main memory and cache size, cache misses, false sharing, hyper-threading [45]. Software limitations includes race conditions, locking, deadlocks, synchronization and granularity of work assigned to a thread.

Different parallel implementations have been tested to better understand and discover non-obvious intricacies in multi-threaded codes. One such problem is false cache sharing [46]. The degrading performance results of Experiment 2.1 brings this issue to light. Also discussed is the impact of cache memory and cache levels that improves parallel performance, but also is the reason for false cache sharing.

In Experiment 2.2, the final result of parallel performance is shown against the sequential execution time obtained in Experiment 2.1.

Experiment 2.1 Trivial Implementation using *N* threads for *N* particles

Particle filters can be classified as a 'trivially parallel problem' as each particle executes or is sampled independently from the other particles. In this experiment, a naïve and direct approach is adopted to execute the code in parallel, by assigning to each of the *N* particles an individual thread of execution from the start of motion model method till the end of the measurement model method. This is in exact adherence to the MCL algorithm explained in Chapter 3. Resampling step is not implemented in parallel, so the algorithm waits for all particles to finish execution of the measurement model method.

Allocating an individual thread per core per particle is not possible in practice because the minimum number of particles required for MCL far outweighs the maximum number of CPUs available in the target processors (i7 or i5) or any other currently available desktops in the market. Executing such a code will result in very high overhead due to the operating system context switching among the *N* threads ($N > 100$), trying to schedule them to run in far lesser number of cores (typically 2,4, or 8). This approach is possible for some high end GPUs as demonstrated in [41][47]. Therefore, though this implementation was performed, the execution time for the lowest particle set (512 particles) was more than 3.5 minutes. Hence, further experimentation was discontinued.

Experiment 2.2 Re-implementation using Parallel Class library.

As discussed in Chapter 5, in order to map such a parallel implementation while avoiding oversubscription or under subscription of available CPU resources, and use dynamic scheduling of work to the threads, the *Parallel Class (Task Parallel Library)* [16] in the .Net framework 4.0 has been used to maintain a pool of threads where each individual thread can continuously process a new particle after finishing its previous execution.

The performance of the experiment is shown in Figure 6-5

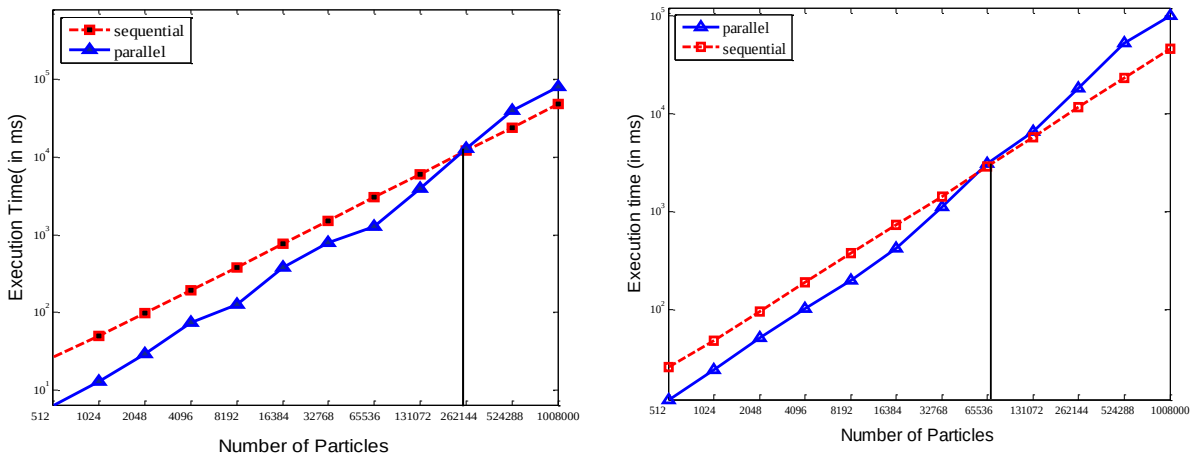
Figure 6-5. As noted in Chapter 5, in this model, there are *n* concurrent threads/tasks executing till all the *N* particles ($1 \leq n \leq \text{maxCPU}$, and maxCPU is the total number of CPUs in the multi-core processor) have been predicted and importance weight calculated. This has been done by using the `Parallel.For()` method in the Parallel Class library of dot net framework 4.0.

One of the advantage of using the class is the avoiding of oversubscription of the CPUs. The number of threads executing concurrently cannot exceed the maximum number of available CPUs thus avoiding the deadlocks and system thrashing along with context switching conditions observed in the previous experiment. Also, the system implicitly creates and maintain a pool of threads that can be re-

used. This not only saves time from thread creation and deletion, it also allows for dynamic sharing of workload, if needed. A similar implementation can also be achieved by using the

S*ystem.Threading.ThreadPool* class for previous versions of dot net framework.

However, it must be noted that the results of Experiment 2.2 indicates poor performance of the parallelized algorithm. The analysis and reasoning is done after the results are presented.



(a) (b)

Figure 6-5 Effect of false sharing and cache size on parallel performance

Figure 6-5 shows the effect of cache size on the parallel performance of the code compared to the sequential implementation. The comparison is made by plotting the execution time of MCL for both implementations. Figure 6-5 (a) shows the result from the i7 processor and Figure 6-5 (b) shows the same from i5.

As seen from the above plots, for lesser number of particles this parallel implementation has total execution time lesser than the sequential version. The blue line (parallel execution time) stays well below the red line (serial execution time) for almost the entire plot. But as the number of particles increases beyond a certain size (marked by the vertical line in the plots), the performance of the parallel code visibly decreases. The performance becomes negative and the parallel implementation takes more time to complete execution than the sequential version.

This negative performance is because of the size of the cache memory. As the total memory required by the code depends on the number of particles, memory misses in the cache starts occurring for larger particle sets when all the data cannot reside in the cache. The memory hierarchy of i7 and i5 consists of two dedicated L1 and L2 and a shared L3 cache. The L3 cache size in i7 and i5 is 6 KB and 3 KB respectively. Therefore, the i5 shows negative performance for a smaller particle set as compared to i7.

Another reason for higher execution time in the parallel implementation is false cache sharing. Figure 6-5 does not accurately portray the adverse effect false sharing could have on parallel performance. This is because the graph is plotted against the average execution time over the entire simulation period. However, the data in Table 6-1 will prove the negative impact of false sharing in multi-threaded applications and in the parallelized MCL.

Execution Time							
PS = 1024		4096		65536		1048576	
Parallel	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel	Sequential
18.28	52.04	66.34	188.19	771.47	2895.81	19325.94	45477.29
18.49	48.45	62.33	183.48	11196.76	2871.76	2946405.47	48132.55
11.72	48.31	55.76	181.89	19541.34	2837.13	2055311.32	45679.84
11.40	49.96	49.74	183.67	674.45	3011.06	10950.78	45835.67
12.37	46.78	53.68	181.18	676.00	2851.59	10648.56	46559.76
12.78	47.06	48.95	180.85	671.46	2852.70	10851.18	46386.62
18.50	46.79	188.41	185.79	3618.24	2850.30	11366.05	45728.91
12.68	47.35	203.81	181.38	10387.36	2843.57	11016.40	46238.82
12.74	47.25	43.84	180.80	1286.91	2842.57	11138.10	45628.32
13.39	46.60	39.38	181.80	690.04	2862.12	11281.68	45755.36
Col(a)		Col(b)		Col(c)		Col(d)	

Table 6-1 Running time of parallel and sequential implementation (Experiment 2.2)

Table 6-1 shows the running time for the first 10 steps of the MCL for the parallel and sequential version with different particle sets. The columns are numbered below in the last row.

Col(a) of the table shows the running time for 2^{10} particles. The performance for this parallel implementation is quite high. Average speedup achieved is approximately **3.37**. As particle size increases, for instance in Col(c) for 2^{16} particles, the parallel implementation takes approximately 695 milliseconds to complete one cycle, but as highlighted, some cycles have very high execution time, about 16 times more

than the average value. For 2^{20} particles, the execution time increases by up to 150 times than the average value of 12,000 milliseconds. This can be attributed to false sharing.

False cache sharing should affect each step of the simulation causing the execution time to increase by a large factor. However, in Table 6-1 only a few steps are affected by false sharing because of the random implementation of `Parallel.For()` method.

False sharing [48] occurs when multiple threads on different CPUs modify variables on the same cache line. Intel multi-core processors like i7 and i5 use the Modified - Exclusive - Shared - Invalid (MESI) protocol to ensure cache coherency of the internal cache lines of each CPU. When one thread modifies a variable in the cache line, the MESI protocol invalidates the entire cache line in the other CPUs to maintain cache coherency. This requires a fetch of the updated cache line from the main memory. Though each thread in each CPU modifies a different variable, yet all the CPUs obtain an updated value of the variable as if they are sharing the same variable for which some locking semantics is required. This is why it is known as false sharing, since no real sharing of variables happen across CPUs or threads.

In the `Parallel.For()` method, the concurrent execution of a number of loop iterations is not executed according to the index numbers or in any orderly manner. For example, Parallel.For method could run the iterations for index 0,16,32,48 in four different threads concurrently instead of 0,1,2, and 3, thus modifying the memory locations of only these indexes which will usually not be present in the same cache line. This out of order execution is carried out in fact to reduce the effect of false sharing by deliberately avoiding writing to contiguous memory locations at the same time.

Similarly, for Experiment 2.2, `Parallel.For()` processes random particles and only updates the corresponding memory locations and hence there is a good average speedup. But since this is random, sometimes it also updates the variables in the same cache line therefore resulting in very high execution time for those steps as seen in Table 6-1.

The results obtained from this parallel implementation of the Monte Carlo Localization also applicable to other particle filter algorithms, has higher significance when performed on cheaper processors with lesser cache memory or clock frequency, typically embedded processors. The Intel i7 and i5 processors used in this research for testing the performance are among the best processors in their category currently available. Therefore it was necessary to test the algorithm with an extremely large number of particles in the order of millions, to accentuate the problems and limitations on the performance of the parallelized implementation of the algorithm.

The main purpose of this experiment was to show **how not to parallelize a particle filter algorithm** and the performance drawbacks a naïve implementation directly derived from the algorithm,

has. This experiment brings out the subtle issues in multi-threading, concerned with the architecture of the underlying processor and memory hierarchy [49], [50]. False sharing is a well-known problem in symmetric multiprocessors (SMPs) and Chip Multiprocessors (CMPs) and has been well studied [46][51].These issues can be easily identified by a seasoned computer architect or parallel programming expert, but it is not an obvious thing to a robotic enthusiast or to researchers in the signal processing community.

The author would also like to state here that the use of the Parallel Class library does not hurt the performance of this algorithm, except in the way it has been implemented. Neither does he discourages its use. In fact, these APIs in the windows platform and similar libraries like Intel's Thread Building Blocks [52], OpenMP [12] [13] tools actually make writing parallel code much easier, though not much obvious.

In the following section, the results of the second parallel implementation are discussed. This model uses the traditional methods of threading concepts along with new versions of synchronizing mechanisms in the DotNet Framework 4.0 to achieve the best possible performance. The relevant details for this experiment has been discussed in in Chapter 4. The run time performance and the speedup for different sets of particles and thread configurations is analyzed and verified.

In Experiment 2.1, it was identified that the motion model and the measurement model takes up the most computation time. Hence, the motivation was to re-implement these modules to execute them in parallel.

Experiment 2.3 Parallelization of the Motion model.

The performance of the parallel implementation of the motion model is plotted in Figure 6-6. The experiment was performed on the i7 processor. The motion model implements the sampling / prediction step of the MCL. The maximum speedup obtained with 8 threads was **3.4**, for 42,000 particles. The execution time of the sequential implementation was used as the baseline for speedup calculation.

An interesting result obtained in this experiment is the negative performance for lower number of particles. It was observed that for less than 4200 particles, implementing parallelism by more number of threads deteriorated the performance of the algorithm.

One of the reasons for this performance issue is the granularity of the task assigned to each thread. With lesser particles, the threads finish execution faster than the time required to communicate with the other threads for data and synchronization. This is known as *parallel overhead* where the thread suffers in performance due to a large number of small tasks. Another reason that could be attributed is the amount of time taken to create a thread, being comparable to the time spent by the thread in execution of the task. This again relates to the granularity of tasks assigned to the threads

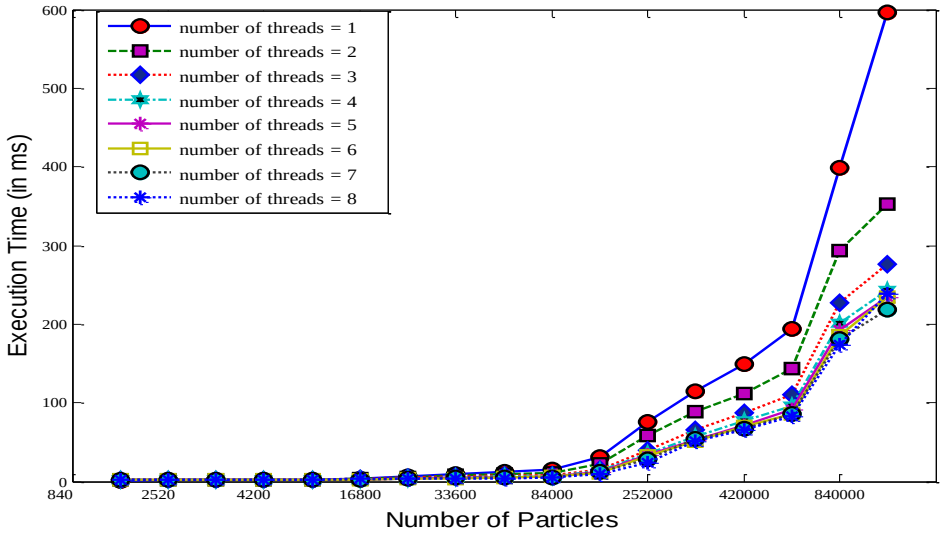


Figure 6-6 Execution Time of motion model implemented in parallel

Besides the hardware issue, it can also be stated that the motion model step is more memory intensive than compute intensive. Therefore the time spend by the threads waiting for data is comparable or higher than the time spent in CPU bound activities. The i7 has a single memory bus to the L3 cache, which is shared by 4 cores and 8 threads.

In order to verify the results obtained from the i7 processor, the same experiment was tested in the i5 processor. The result obtained were in fact better than the results from the i7 for particles lesser than 4,200. In i5 the workload (number of particles) is divided by 4 for the four concurrent threads as compared to 8 threads in the i7, resulting in more work per thread in the i5.

Experiment 2.4 Parallelization of the Measurement Model

The measurement model executes for more than 97% of the total computation time of MCL. Measurement model implements the state update method. Paralleling this step will have significant performance gain in the MCL.

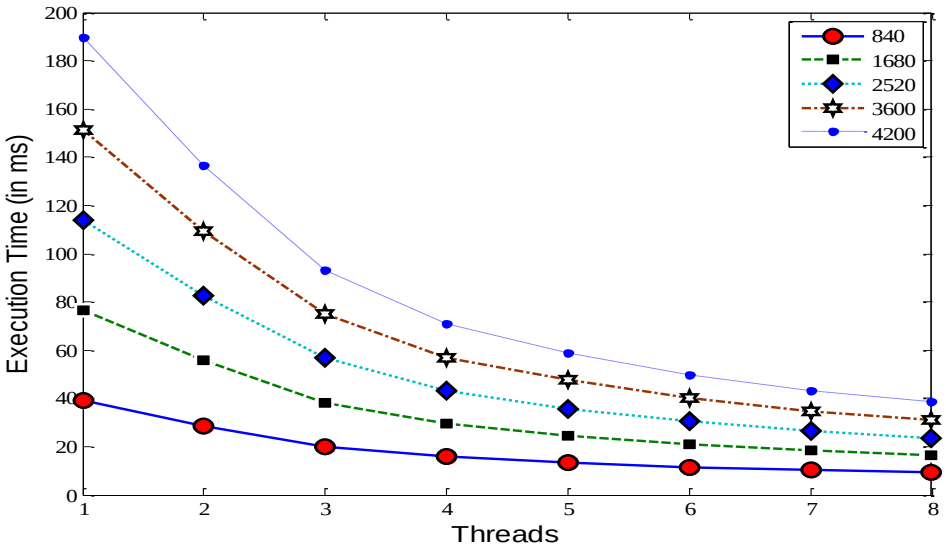


Figure 6-7 Execution Time of measurement model implemented in parallel

Figure 6-7 plots the execution time (in milliseconds) of the measurement model verses the number of threads. The negative slope of the curves indicates a reduction in the execution time as the number of threads increases.

An observation similar to the motion model step was made on the number of particles used and the total performance gain. It can be verified from the above figure that the rate of decrease in computation time for 4,200 particles is significantly higher than that for only 800 particles.

From the results of both Experiment 2.3 and 2.4 it can be suggested that, lesser the work a thread does, lesser is the gain it receives from parallelism.

Another observation made is the non-linear performance associated with threading and parallelism. In the sequential implementation, as shown in Experiment 1.1 and 1.2, the performance of the algorithm scales linearly as the number of particles increases. However, the plots in Figure 6.6 and Figure 6.7 reveals a non-linear relationship.

Experiment 2.5 Performance of the Parallel Monte Carlo Localization algorithm.

This experiment discusses the results on the performance of the complete parallelized implementation of MCL. The speed up gained with threading is relative to the sequential single threaded implementation of the algorithm.

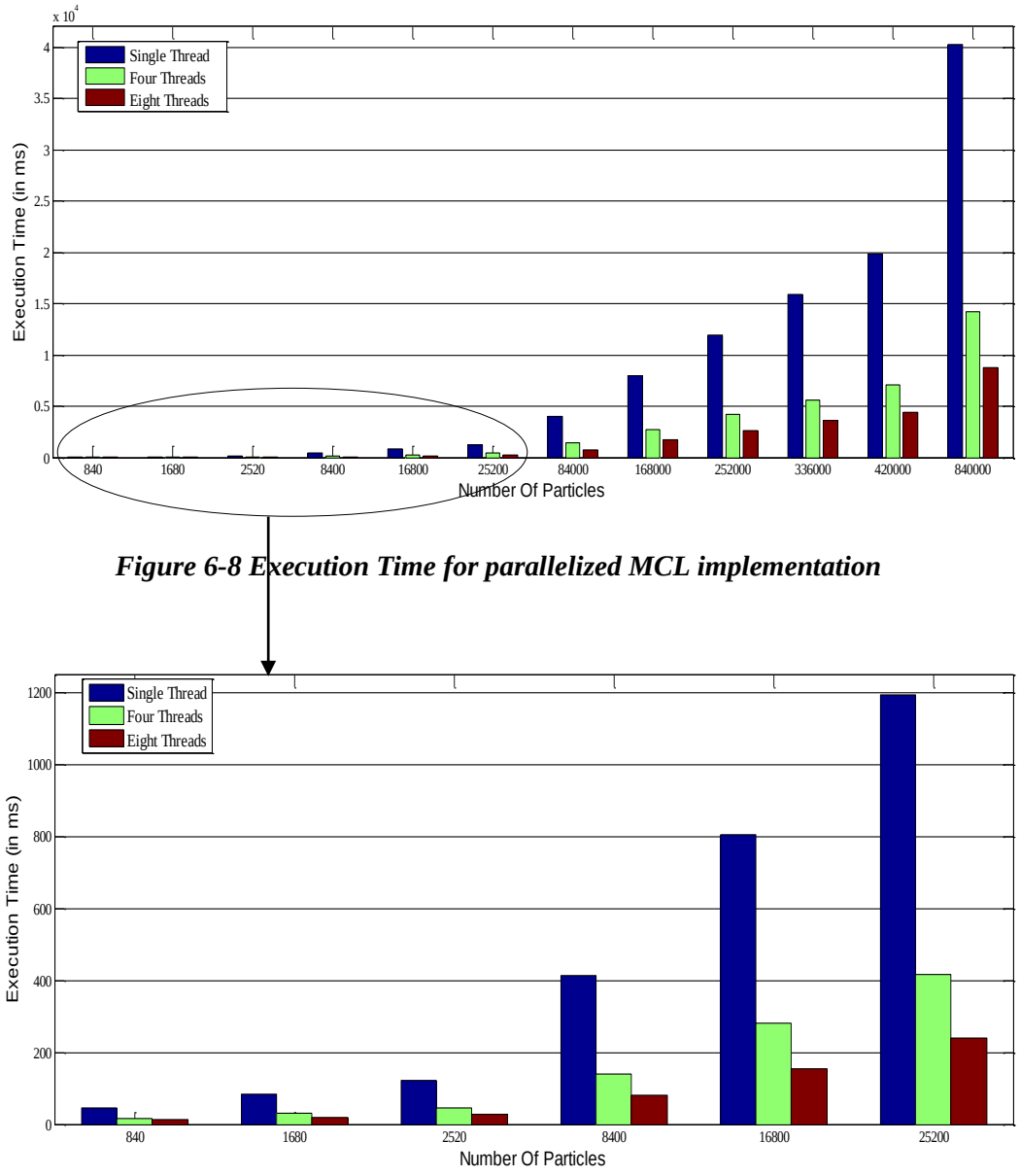


Figure 6-8 Execution Time for parallelized MCL implementation

Figure 6-8 plots the execution time of the multi-threaded implementation of MCL, for one, four and eight threads. The run-time was profiled on the i7, with a maximum of 8 threads.

The number of particles are selected such that there is a linear as well as a logarithmic increase in the numbers. This is to exemplify the speed up that was gained from multi-threading. It can be seen from Figure 6-, the execution time of 25,200 particles with only 4 threads is equal to that of a sequential implementation with 8,400 particles. Similarly, the 8-threaded implementation with 840,000 particles has a similar execution time to 168,000 particles in sequential. This is plotted in Figure 6-.

The results obtained were optimistic. They show that with multi-threading the inherent data-parallelism observed in a particle filter implementation can be exploited in a multi-core processor capable of running multiple operations concurrently. The maximum speed up obtained was **5.43**.

However, the speedup achieved is not linear with the number of threads. That is, by using 8 threads to parallelize an algorithm does not give a speedup of 8 times. Infact, the maximum speedup is quite less than 8. This is shown in Figure 6-.

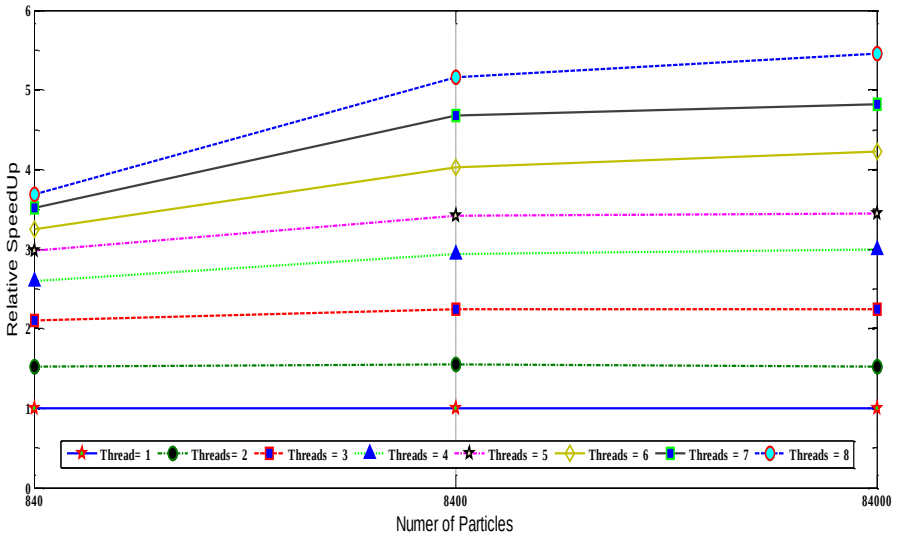


Figure 6-9 Relative speedup with 8 threads

Figure 6- compares the relative speed up gained with 8 threads for three different particle sets. The single threaded version is the base line, for which the speedup is always 1.

It can be observed from the plot the maximum speed up attained for any particle set is less than 8 throughout the plot. Also observed is the discontinuity in increase in speedup as the number of particles increases from 840 to 84,000. The change in slope of the linear plot also proves that the speed up that can be achieved by multi-threading has hardware and software limitations as discussed earlier.

Comparison with Amdahl's law

The speedup obtained from the parallelized implementation of the MCL is comparable to the theoretical speedup predicted by Amdahl's law [19] which proposes a theoretical maximum speedup that can be achieved by parallelizing an algorithm.

The parallelized implementation of MCL is highly suitable for Amdahl's speed up equation as the MCL also has a section of serial execution in the resampling step. The speed up equation is a function of the fraction of the serialized portion of the code for a given data size.

Table 6-2 shows the speed up calculated from Amdahl's law speed up equation against the actual speed up achieved for 84,000 particles for 2-8 threads.

Threads	Amdahl's law speedup	Experimental speedup
2	1.972	1.527
3	2.882	2.240
4	3.710	2.850
5	4.469	3.448
6	5.126	4.244
7	5.705	4.931
8	5.935	5.431

Table 6-2 Speedup from Amdahl's law

The maximum speed up predicted by Amdahl's law was 5.935 for 8 threads, and the speed up obtained from the parallel implementation was **5.413**.

Amdahl's law is a theoretical approximation for maximum speed up calculation. It does not take into considerations overheads such as communication, synchronization, thread management and in this case, hyper threading- where 8 threads can run concurrently in 4 physical cores, this requires sharing of hardware resources like cache between the two threads running on a single CPU.

CHAPTER 7

CONCLUSION

The results obtained from all the experiments justifies the basis of this research. A particle filter algorithm can be designed to take advantage of a multi-core processor. This was tested and verified on a quad core Intel i7 processor and a duo core Intel i5 processor.

The MCL algorithm which is an example of the SIR particle filter was studied in details and was implemented. A robotic simulator was designed to generate measurement and control data. The accuracy of the simulator was also verified, in which MCL was able to globally and locally localize the pose of a mobile robot.

The performance of the code was evaluated in both the i7 and i5 processors, which proved that a sequential implementation cannot take advantage of the available parallelism in hardware or the parallelism in the algorithm. Therefore the algorithm was redesigned and parallelism was implemented by adding threads to sections of the code, which were identified to be parallelizable. This is known as multithreading. The results of the tests showed that for 8 threads running concurrently, a speedup up to **5.43** times can be achieved in the parallel implementation. For many applications, such speedups can meet real time requirements.

The level of parallelism was varied by changing the number of threads in the code. The threads were also allocated specific CPUs, which eliminates thread migration and jumps among CPUs and also improves cache performance by taking advantage of data locality. The hardware limitations on achieving parallelism by threading were also identified.

APPENDIX

Processor Configuration

Name	Intel Core i7 2630QM	Intel Core i5 2410M
Clock Speed	2.00 Ghz	2.30 Ghz
No of Cores	4	2
No of Threads	8	4
Memory	8 GBytes	4 GBytes
Cache		
L1 Data	4 x 32 KBytes	2 x 32 KBytes
L1 Instruction	4 x 32 KBytes	2 x 32 KBytes
L2	4 x 256 KBytes	2 x 256 KBytes
L3	6 MBytes	3 MBytes

REFERENCES

[1] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "A novel approach to nonlinear / non-Gaussian Bayesian state estimation," *IEE Proceedings F Radar and Signal Processing*, vol. 140. p. 107, 1993.

[2] M. Isard and A. Blake, "Condensation-conditional density propagation for visual tracking," *Int. J. Comput. Vis.*, vol. 29, no. 1, pp. 5–28, 1998.

[3] D. Fox, W. Burgard, F. Dellaert, and S. Thrun, "Monte Carlo Localization: Efficient Position Estimation for Mobile Robots," *Proc. Natl. Conf. Artif. Intell.*, pp. 343–349, 1999.

[4] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit, "FastSLAM 2.0: An Improved Particle Filtering Algorithm for Simultaneous Localization and Mapping that Provably Converges," *Int. Jt. Conf. Artif. Intell.*, vol. 18, pp. 1151–1156, 2003.

[5] V. Verma, G. Gordon, R. Simmons, and S. Thrun, "Particle Filters for Rover Fault Diagnosis," no. June, 2004.

[6] M. D. Breitenstein, F. Reichlin, B. Leibe, E. Koller-Meier, and L. Van Gool, "Robust tracking-by-detection using a detector confidence particle filter," *Comput. Vision, 2009 IEEE 12th Int. Conf.*, 2009.

[7] A. U. Peker, O. Tosun, and T. Acarman, "Particle filter vehicle localization and map-matching using map topology," *2011 IEEE Intell. Veh. Symp.*, pp. 248–253, 2011.

[8] F. Gustafsson, F. Gunnarsson, N. Bergman, U. Forssell, J. Jansson, R. Karlsson, and P.-J. Nordlund, "Particle filters for positioning, navigation, and tracking," *IEEE Trans. Signal Process.*, vol. 50, 2002.

[9] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking," *IEEE Trans. Signal Process.*, vol. 50, 2002.

[10] S. Borkar, "Thousand Core Chips: A Technology Perspective," *Proceedings of the 44th annual Design Automation Conference*, pp. 746–749, 2007.

[11] J. Parkhurst, J. Darringer, and B. Grundmann, "From Single Core to Multi-Core: Preparing for a new exponential," *2006 IEEE/ACM Int. Conf. Comput. Aided Des.*, 2006.

[12] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, pp. 46–55, 1998.

[13] "OpenMP." http://www.openmp.org/.

[14] Microsoft, “Parallel Programming in the .NET Framework.” http://msdn.microsoft.com/en-us/library/ff963553.aspx.

[15] P. S. Pacheco, “Parallel Programming with MPI,” *Perform. Comput.*, pp. 1–43, 1997.

[16] Microsoft, “Task Parallel Library (TPL).” http://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx.

[17] I. M. B. Nielsen and C. L. Janssen, “Multi-threading: a new dimension to massively parallel scientific computation,” *Comput. Phys. Commun.*, vol. 128, pp. 238–244, 2000.

[18] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995.

[19] G. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the AFIPS spring joint computer conference SE - AFIPS ’67 (Spring)*, 1967, pp. 483–485.

[20] S. Thrun, B. Wolfram, and F. Dieter, *Probabilistic Robotics*. 2005.

[21] A. Doucet, N. De Freitas, and N. Gordon, “Sequential Monte Carlo Methods in Practice,” *Springer New York*, pp. 178–195, 2001.

[22] J. S. Liu and R. Chen, “Sequential Monte Carlo Methods for Dynamic Systems,” *J. Am. Stat. Assoc.*, vol. 93, pp. 1032–1044, 1998.

[23] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Trans ASME J Basic Eng*, vol. 82, pp. 35–45, 1960.

[24] G. Welch and G. Bishop, “An Introduction to the Kalman Filter,” *In Pract.*, vol. 7, pp. 1–16, 2006.

[25] M. K. Pitt and N. Shephard, “Filtering via simulation: Auxiliary particle filters,” *J. Am. Stat. Assoc.*, vol. 94, pp. 590–599, 1999.

[26] A. Doucet, N. de Freitas, K. P. Murphy, and S. J. Russell, “Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks,” in *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, 2000, pp. 176–183.

[27] G. Kitagawa, “Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models,” *J. Comput. Graph. Stat.*, vol. 5, pp. 1–25, 1996.

[28] J. J. Leonard and H. F. Durrant-Whyte, “Mobile robot localization by tracking geometric beacons,” *IEEE Trans. Robot. Autom.*, vol. 7, 1991.

[29] J. Borenstein and Y. Koren, “Obstacle avoidance with ultrasonic sensors,” *IEEE J. Robot. Autom.*, vol. 4, 1988.

[30] F. Dellaert, W. Burgard, D. Fox, and S. Thrun, “Using the CONDENSATION algorithm for robust, vision-based mobile robot localization,” *Proceedings. 1999 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (Cat. No PR00149)*, vol. 2, 1999.

[31] S. Se, D. Lowe, and J. Little, “Vision-based Mobile Robot Localization And Mapping using Scale-Invariant Features,” in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, 2001, pp. 2051 – 2058 vol.2.

[32] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, “Robust Monte Carlo localization for mobile robots,” *Artif. Intell.*, vol. 128, pp. 99–141, 2001.

[33] J. E. Handschin, “Monte Carlo techniques for prediction and filtering of non-linear stochastic processes,” *Automatica*, vol. 6. pp. 555–563, 1970.

[34] W. D. Hillis and G. L. Steele, “Data parallel algorithms,” *Communications of the ACM*, vol. 29. pp. 1170–1183, 1986.

[35] J. Subhlok and G. Vondran, “Optimal Mapping of Sequences of Data Parallel Tasks,” vol. 1, no. July, 1995.

[36] H. Durrant-Whyte and T. Bailey, “Simultaneous localization and mapping (SLAM): part I The Essential Algorithms,” *Robot. Autom. Mag.*, vol. 2, pp. 99–110, 2006.

[37] S. Toub, “Patterns of parallel programming,”. 2004.

[38] I. Corporation, “Developing Multithreaded Applications : A Platform Consistent Approach,” Intel Corporation, 2003.

[39] "*Intel Guide for Developing Multithreaded Application"*. Intel Corporation, 2011.

[40] G. Hendeby, R. Karlsson, and F. Gustafsson, “Particle Filtering: The Need for Speed,” *EURASIP Journal on Advances in Signal Processing*, vol. 2010. p. 181403, 2010.

[41] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron, “Parallelization of Particle Filter Algorithms,” 2010.

[42] G. E. P. Box and Mervin E. Muller, “A Note on the Generation of Random Normal Deviates,” *Ann. Math. Stat.*, vol. 29(2), pp. 610–611, 1958.

[43] Microsoft, Lambda Expressions (C# Programming Guide) http://msdn.microsoft.com/en-us/library/bb397687.aspx.

[44] J. L. Gustafson, “Reevaluating Amdahl’s law,” *Communications of the ACM*, vol. 31. pp. 532–533, 1988.

[45] “Intel Hyper-Threading Technology,” January, 2003.

[46] D. Chandler, “Reduce False Sharing in .NET,” *Intel Corporation*.April, 2005

[47] K. Par and O. Tosun, “Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures,” in *2011 IEEE Intelligent Vehicles Symposium (IV)*, 2011, no. Iv, pp. 820–826.

[48] J. Torrellas, H. S. Lam, and J. L. Hennessy, “False sharing and spatial locality in multiprocessor caches,” *IEEE Trans. Comput.*, vol. 43, 1994.

[49] J. C. Mogul and A. Borg, “The effect of context switches on cache performance,” *ACM SIGARCH Comput. Archit. News*, vol. 19, pp. 75–84, 1991.

[50] J. B. Chen and B. N. Bershad, “The impact of operating system structure on memory system performance,” in *ACM Symposium on Operating Systems Principles*, 1994, pp. 120 – 133.

[51] T. E. Jeremiassen and S. J. Eggers, “Reducing false sharing on shared memory multiprocessors through compile time data transformations,” *ACM SIGPLAN Notices*, vol. 30. pp. 179–188, 1995.

[52] “Intel Threading Building Blocks.” https://www.threadingbuildingblocks.org.